The Pragmatic Programmers

Broadview®
www.broadview.com.cn

Serverless Single Page Apps
Fast, Scalable, and Available

Serverless架构
无服务器单页应用开发

[美] Ben Rady 著
郑美赞 简传挺 译

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Serverless Single Page Apps

Fast, Scalable, and Available

Serverless架构

无服务器单页应用开发

[美] Ben Rady 著

郑美赞 简传挺 译

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书讲授如何利用 Amazon 公司的 AWS Lambda 创建 Serverless 单页应用。这里，Serverless 的意思是应用开发者无须管理服务器，将应用构建在服务之上，而不是运行在需要人工配置和维护的服务器之上。这种新的开发方式带来很多好处，比如节省成本，可扩展性与可靠性高，以及开发者可以专注于实现应用的业务逻辑等。全书共 8 章，Ben Rady 带领读者采用这种新方法从零开始开发一个 JavaScript 解题应用，并且对其进行测试，最终完成部署。

对于创业者以及中小企业的开发者来说，本书讲述的 Serverless 设计是一个值得了解和学习的新方法，可以从中获得启示，抓住先机。

版权贸易合同登记号 图字：01-2016-7931

图书在版编目（CIP）数据

Serverless 架构：无服务器单页应用开发 /（美）本·雷迪（Ben Rady）著；郑美赞，简传挺译. —北京：电子工业出版社，2017.7
书名原文：Serverless Single Page Apps:Fast,Scalable,and Available
ISBN 978-7-121-31736-1

Ⅰ. ①S… Ⅱ. ①本… ②郑… ③简… Ⅲ. ①移动终端－应用程序－程序设计 Ⅳ. ①TN929.53

中国版本图书馆 CIP 数据核字(2017)第 124192 号

责任编辑：许 艳
印 刷：北京中新伟业印刷有限公司
装 订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编：100036
开 本：787×980 1/16 印张：14 字数：227.4 千字
版 次：2017 年 7 月第 1 版
印 次：2017 年 7 月第 1 次印刷
定 价：65.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

本书赞誉

软件行业里聚集着最多的精英——上百万的开发者，他们带动技术朝着代码更容易测试、解决方案更简单、结果更可靠，以及维护起来更轻松的方向发展。有人看到了 Serverless 设计的未来，然后回过头来教授我们这些后知后觉者如何开发下一代应用，Ben 就是这样的开拓者。他的书就像是一位循循善诱的老师，教你理解 Serverless 设计模式，引导你自然地遵守部署和测试的最佳实践。

Tim Wagner

@timallenwagner

本书对于所有背景的开发者来说都是一份翔实而通俗易懂的指南。不管你是否使用 AWS，都能学到不少知识——从应用的安全到访问数据时不可或缺的身份认证。

Will Gaul

Ben 在本书中讲了很多内容：用 JavaScript 构建客户端逻辑、用 Cognito 进行认证和授权、用 Lambda 实现不能放心地交给浏览器处理的敏感功能。JavaScript 开发者会从中发现一些实现典型服务端功能的新方法，而且读完本书，你就会得到一个成本近乎为零的能运行的 Serverless 应用。

Ryan Scott Brown
serverlesscode.com 的作者
Serverless Framework 贡献者

未来你的应用不再运行在应用服务器上——而是运行在你公司某个机柜的机器里，运行在云上，由一组可靠的服务保护和管理。跟着本书开启全新的开发之旅吧！

Daniel Hinojosa
Testing in Scala 作者

本书对 Serverless Web 应用开发这种前沿技术做了精彩的介绍。它将带着你从零开始，直到部署 Serverless 应用。

Jake McCrary
Outpace Systems 公司软件开发主管

我读过很多技术图书，这一本是我今年读过的最好的书，也是我这些年读过的最好的书之一。Ben Rady 的讲述既轻松又实在，没有吹嘘自己的知识，也没有用不必要的内容凑篇幅。书中不仅告诉你要做什么，而且解释了为什么这么做，两者并重，十分清楚明了。Ben 的观点和技术选型有理有据，非常靠谱。建议你阅读本书。

David Rupp
RuppWorks LLC

谨以此书献给我的家人：

总是赐予我力量的妻子 Jenny

善良的女儿 Katherine

梦想着拯救世界的儿子 Will

致　谢

感谢我的编辑 Jackie Carter，以及 Dave、Andy 和 Pragmatic Bookshelf 的所有员工，感谢他们付出的时间，给予的支持和能量。没有你们，我不可能写这本书。

感谢我的技术审稿人，包括 Alex Disney、Cliton Begin、Daniel Hinojosa、David Rupp、Jake MacCrary、James Ross、Jeff Sacks、Joshua Graham、Lucas Ward、Rene Duquesnoy、Rob Churchwell、Ryan Brown 和 Sebastian Krueger。

感谢 Amazon 所有帮助我验证本书中想法的人，包括 Tim Wagner、Bob Kinney、Tim Hunt 和 Will Gaul。

另外，感谢所有在本书 beta 版本期间提供反馈的人，不管是私下还是通过勘误页面，是在论坛上还是在 Github.com 上提出意见的人，包括 Bill Caputo、Christopher Brackert、Christopher Mowller、Dennis Bruke、Ezequiel Rangel、Fred Daoud、Hal Wine、Jerry Tang、Ron Campbell 和 Timm Knape。

感谢所有帮助我写这本书的人！你们的反馈对我来说非常宝贵，真心感谢你们付出的时间和精力。

前　言

我做了几年单页 Web 应用，一直希望能摆脱应用服务器施加的限制，现在愿望终于实现了。Amazon（还会有更多的其他公司）开发出的技术让无服务器架构（serverless，在本书中简称为“无服”）成为可能，消除了很多构建和扩展 Web 应用的风险和成本。这个创意是如此令人叹服，如此具有变革意义，以至于我必须为它写一本书。

这本书是为那些希望在 Web 上构建一些东西的人而写的。有的人想搭建一个应用——一个他们认为将会是下一个惊世之作的东西。有的人只是刚着手 Web 开发，从未构建过任何类型的应用——不管是影响世界的还是其他什么样的。有的人则是已经用 Java、Ruby on Rails 或者 Node.js 构建过许多基于 Model-View-Controller 应用的资深 Web 开发者。随着这个无服技术的兴起，我希望能分享所学的一切，真心希望有人能够用它做出一些了不起的东西。

在前言中，你将了解本书的主题以及能用这些技术来实现些什么。

指导原则

写这本书时我脑子里有几条指导原则。这些原则的目的是让本书内容具有针对性、易

理解，不说废话。这里把它们列出来，帮助你了解我是如何写这本书的以及更好地理解本书的相关背景。

有些原则可能是有争议的，其中一些甚至与构建 Web 应用的主流思想相左。但是，这些原则将帮助你深入理解这个主题。与其波澜不惊，存在争议会更好。

使用 Web 标准和熟悉的工具

在这本书中，你将使用一小部分精选的工具来构建一个单页 Web 应用。在本书的某些节点上，你将会实现其他工具已有的功能，这些工具是我有意不在应用中引入的。你可能会好奇，既然它们提供了所需的功能，为什么我们不用呢？

阅读本书实际上是一个学习的过程。当学习知识时，使用熟悉的工具是有帮助的。不然，花在学习工具上的时间比花在学习技术上的还要多。我不希望对一个框架或者库的选择让我们偏离本书主题。这本书是关于无服 Web 应用，而不是关于框架或者类库的。为了保持未知性，我们将会使用那些 Web 开发者熟悉的工具（例如 jQuery）、Web 标准和 Web 服务，来实现一个无服设计。

这本书是关于 Web 服务，而不是 Web 框架的。

有可能读完这本书，你将会接触到一种客户端 Web 框架，比如 React 或者 Angular，来构建你自己的 Web 应用。这些工具近些年在 Web 开发者社区获得了很多关注，我希望能看到很多使用它们的成功项目。在本书中学到的所有知识与你希望使用这些框架做的事情都是兼容的。它们是互为补充而不是相互冲突的关系。

使用函数式 JavaScript

本书中你将不会创建 JavaScript 的任何类。创建类结构来解决问题对于一些具有富类型和面向对象类型系统的语言是合理的，但 JavaScript 不是这样的语言。相反，我们将使用更易于理解的函数式风格。

这意味着你将不会遭遇 this 关键字的范围问题。你将会避免原型和继承的共同使用。你不会使用 new 关键字和专门设计的函数来创建对象；相反，你将会使用一个字面意义的对象来创建它们：{}。

你可以自己决定这是否就是你希望采纳的风格。因为这种方式拥有一些实际、立竿见影的好处，最后许多软件设计决策实际上都会偏向这种风格。代码终究应该首先是给人写的，然后才是计算机。只要它可执行，对于计算机而言代码看起来如何无所谓。

避免无用功

做项目时，我的目标一直都是：持续交付有改进的结果，只要这个改进是朝着环境需要的方向在进行。实现这个目标意味着要避免任何能造成交付率被消磨的事情，比如剃牦牛（yak shaving）。

如果你对“剃牦牛”这个词不熟悉的话，想象一下你想要给朋友买一件毛衣。你走进一家商店却发现没有毛衣卖。幸运的是，那里的另一位客户知道街角有一个很好的裁缝，他可能可以为你做一件。所以你到了裁缝那儿，他有一个非常好看的毛衣图案和一台能编织这个图案的机器，但毛线供应商今天还没送货。所以你跑到供应商那儿……

然后这个过程一直继续，直到你发现自己在西藏的一片牧场上剃耗牛毛来编毛线。“我怎么来这里了？”你可能会问自己，“我所需要的只是一件毛衣。”当一系列看起来相关和必要的任务让你从真实目标上偏离时，你就是在剃耗牛。幸运的是，剃耗牛的解决办法很简单，就是意识到你自己一直在剃耗牛，然后转而买一顶帽子。

我希望你在阅读本书时不要剃耗牛。这就是为什么我用了一个预备好的工作空间以及尽可能少的工具的原因。你应该把时间花在学习上，而不是安装软件、配置和排错上。

通过测试来加快进度

你有没有曾经害怕过修改甚至只是修改一点点代码？也许你当时不确定应该做什么或者为什么应该做这个。“首先，不作恶”对程序员和医生而言均适用。这种情况会让你感

觉进退两难。

设想你做这些改动时，有一位可信的同事——这类系统的专家——就坐在你身边。如果你引入任何缺陷到系统中，他就会拦住你，清楚而明确地告诉你为什么这个改动是个坏主意。如果你有这样可信的同事，是否还会进退两难？

不确定性拖慢了我们的脚本，并且限制了解决方案的范围。为了快速搭建一个软件，你必须有自信。为了获得这份自信，你可以为自己创造一个自动化专家——它了解系统的所有细节，知道系统如何运转以及为什么这样运转。这位专家和系统相辅相成，随着它的改变而改变，互相影响。这个专家就是你编写一套测试，它们快到能持续运行，每秒能运行成百上千的测试，而且每次在你对代码进行修改之后都是如此。

一旦你有了测试套件，就拥有了新选择。而当你不再害怕修改代码时，就可以按照自己的设想来设计应用，而不是试图让它在一开始就要“正确”。这意味着你可以随着形势的变化快速适应这个世界，而不是试图预测你可能需要和不需要什么。你可以基于当下的信息而不是靠猜来做决策，应用就会自然而然地变成它应该的那个样子。

在本书中，你将使用测试驱动开发（TDD）技术来编写应用和它的测试。了解如何用测试来构建软件是一项技能，要获得它的益处，就必须真正地使用它。我在本书中引入 TDD 的目的不仅是向你展示如何测试 Web 应用的特定功能，还想证明用它来测试典型 Web 应用有多简单，如果你知道如何实现的话。

在实践 TDD 的过程中，有一个三步循环，经常描述为红—绿—重构。首先写一个测试，检查应用尚不具备的功能。如果测试如你预期的那样失败了，就可以相信它是在测试你希望加到应用上的功能。现在这些测试是红色的。一旦有一个失败的测试，添加几条最简单的代码来让这个测试通过，通常几行代码即可。现在这些测试是绿色的。

通过测试为应用添加一些功能后，就该后退一步，让自己看得更全面一些。是否在实现中引入了一些重复代码？所有的变量和函数是否都有描述性的准确名字？是否还有更简单的方式？考虑这些事情，然后开始重构。重构是在不改变功能的情况下修改代码。你之

前编写的测试会告诉你是否改变了功能，所以只能在这些测试都通过时重构，这一点很重要。有了它们作为你的安全防护网，可以在开始下一步测试之前清理完所有代码的问题。

TDD 用得越多，你的进展就会越快，从中获得的价值就越多。通过一遍又一遍地重复“红—绿—重构”过程，你将学会如何渐进式地构建一个设计合理、测试充分的应用。这样你不仅对自己的应用有信心，而且对它进行持续的修改也更容易。

边做边学

本书是一本教程，所以你可以跟着书中的引导边做边学。通过这本书，你将构建一个无服单页 Web 应用。本书的目的是用具体的方式解释无服架构的原理。本书的成果是一个可运行的应用[1]，所以你大可以放心，书中的技术就像广告中说的那样有用。

采用这种方式意味着篇幅有限，我对有些知识点不可能讲得很深入。因此，我在每章中安排了一节“下一步”来提供一些主题，如果你希望了解更多，可以查找相关的资料。

从预备好的工作空间开始

为了让你能快速上手，我提供了一个预备好的工作空间（prepared workspace），里面包括了起步需要的所有东西，并且不需要太多时间来设置。这就好比你在画一幅艺术品，我已经为你准备好颜料、画架和画布。你要做的就是创作。

为了能使用这个预备好的工作空间，你需要一台兼容 Bash shell 的计算机来使用工作空间里面的脚本和工具。可以是 OS X、任何*nix 版本或者 FreeBSD。如果你安装了 Cygwin，用 Windows 可能也行。你还需要一个带开发者控制台的 Web 浏览器。本书的大部分例子中使用的是 Google Chrome，但大多数情况下 Firefox 也能提供了类似的功能。

---

1 http://learnjs.benrady.com

如何阅读本书

阅读一本书可以有很多种方式，不仅是从头到尾的那种。至于如何阅读本书，取决于你想要获得什么。下面列出的是读本书的一些常见理由，以及我对于如何使用这本书来实现这些目标的建议。

目标 1：理解无服和传统单页应用的优劣

需要做什么

1. 阅读第 1 章的前 3 节理解两种方式的优劣。
2. 略读第 1 章剩下的部分和第 2～3 章。
3. 通读第 4～8 章，尽可能跟着教程实现应用。

如果你是一个有经验的 Web 开发者，搭建过单页 Web 应用，希望了解更多有关无服 Web 应用的知识，你可能不需要看前 3 章的所有内容。这些章节展示了搭建单页 Web 应用的 vanilla.js 方式。其意图是阐述一个单页 Web 应用的基础模块。我尽量定义哪些部分是必需的，并提供一些可供参考的基础实现。如果你没有给 Web 应用编写过很多测试，可以在第 2 章中实现一些测试例子来学习一些新技能。

读完第 1 章的前 3 节，你会希望集中精力细读第 4～8 章。我建议你搭建一个简单 Web 应用，或者至少一个原型，这样你可以在后面的章节中自己尝试这些技术。通过实验可以学到很多。

目标 2：搭建你的第一个单页应用

需要做什么

1. 在开始前，先读一读书中列出的必要的辅助阅读材料。
2. 从头到尾读完本书所有的章节，如果有必要，附录也要看。

如果你是第一次搭建 Web 应用，你会希望阅读整本书，从头到尾读一遍。另外，你可能也想跟上基础 Web 技术的脚步，包含 HTML、CSS 和 JavaScript，可以查阅下面的免费资源。等理解了这些主题，再来深入读本书。

免费资源

Learn to Code HTML & CSS [How14]，Shay Howe 著[2]

Eloquent JavaScript [Hav14]，Marijin Haverbeke 著[3]

目标 3：使用你喜欢的 Web 框架搭建一个无服应用

需要做什么

1. 通读第 1 章。
2. 用你喜欢的 Web 框架实现第 2 章和第 3 章的测试与功能。
3. 通读第 4～8 章。

正如前面说的那样，我不希望把这本书的重点放在 Web 框架上，但还是可以使用它们来搭建无服单页应用。如果你对单页应用的基本元素很熟悉，并且靠客户端 Web 框架来提供这些功能，可以用喜欢的框架轻松重建我们在第 2 章和第 3 章实现的功能。一旦有了这个作为基础，应该可以遵照本书余下的部分，用这个框架实现整个应用。

目标 4：创建一个最小可行产品（Minimum Viable Product，MVP）

需要做什么

1. 阅读第 1 章的前 3 节，理解无服开发与传统开发方式的优劣。
2. 阅读第 8 章来了解用这种方式构建一个 MVP 的成本。
3. 如果这个方式看起来行得通，阅读剩余的所有章节（如果有需要的话，

2 http://learn.shayhowe.com/html-css/

3 http://eloquentjavascript.net/

包括附录）。

3. 使用预备好的工作空间作为搭建 MVP 的起点。

当启动一个新产品或者一项新业务时，首要的挑战是搞清楚市场想要什么以及它愿意为什么付钱。许多人称之为产品/市场匹配（product/market fit），这是构建一个成功产品或者服务的关键。

找到产品/市场匹配的一个有效方式是先做出一个产品并试着销售。验证市场的需求和你通过销售或市场渠道连接这些客户的能力，这是你应该尽快跨过的关键门坎。当然，你可能又没那么多时间和资金来搭建一个完整的应用，所以一个替代方案是搭建一个 MVP 来验证产品的核心价值。

无服单页应用是尝试新主意、探索可能的新市场或者创建 MVP 的好方式。搭建这类应用来替代传统 Web 应用或者原生应用，意味着你可以更快接触到客户。你可以在若干小时内搭建一个初始版本并在几秒内部署。这些应用可以立即更新，轻松地被拆分测试，可以提供详细的使用指标，帮助你理解客户想要的是什么。

除了运行起来不贵、快速构建以及几乎所有用户都能访问之外，无服单页应用可以“无限”（正如 Amazon 宣传的那样）扩展，这样如果你的产品在市场上有旺盛的需求，你就可以满足这样的需求以及留住用户。

在线资源

你可以在 Progmatic Bookshelf 官网找到这本书中的应用以及代码[4]。你还能在上面看到社区论坛和提交勘误的表单，报告问题或者为未来的版本提供建议。

4 https://pragprog.com/book/brapps/serverless-page-apps

你可以在我的 Github.com 账号（benrady）[5]下找到预备好的工作空间。如果你还没有 Github 账号，那就新建一个，并且 fork 我的代码库。想知道更多操作细节，可以阅读第 1 章的内容。

Ben Rady
benrady@gmail.com

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误**：您对书中内容的修改意见可在 *提交勘误* 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 *读者评论* 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/31736

5 https://github.com/benrady/learnjs

目　录

第 1 章
从简单开始

如果你曾经想过“应该有一个实现这种功能的应用”，并憧憬有谁能够为你开发一个就好了，现在我们有一个好消息，那个人找到了，就是你自己。

Web 应用可以是非常强大、高效和易扩展的，但却不应该是复杂的。简单就是 Web 应用的一大优势。你可以利用这种优势来搭建自己的解决方案，实现自己的创意。一旦了解所有模块是如何搭建到一起的，你就能开发出想要的应用了。

本书是一本实用教程，将会演示一种无服务器的方案来搭建 Web 应用。使用这个方案，大部分运维方面的问题就不需要你自己操心了，而且也省去运行服务器的费用。你能集中时间和精力开发想要的应用，而让其他人去考虑业务发展带来的应用上线、配置、升级、扩展服务器等问题。使用多层 Web 框架、自动生成的代码或者拷贝模板，这些方法并不会带来这样的好处。等我们一起学完本书，你就会知道如何通过移除部分代码和消除中间层来交付更好的应用。

为了能够快速演示开发过程，我们将使用一个预设好的工作空间，里面加载了搭建完整 Web 应用必需的所有模块。首先，我们会完成一个单页应用，用 JavaScript、HTML 和 CSS 代码来实现原来在服务器端实现的逻辑。我们将根据 Web 标准，深入挖掘单页 Web

应用的必要功能，从零开始搭建，从而了解它们的运行机制，保证这种设计能符合我们应用的要求。当仅凭 Web 标准不能完全实现需求时，我们会使用 jQuery 来填补差距。最后，我们会使用测试优先的方法来渐进式开发，以保证单页应用的可测试性。

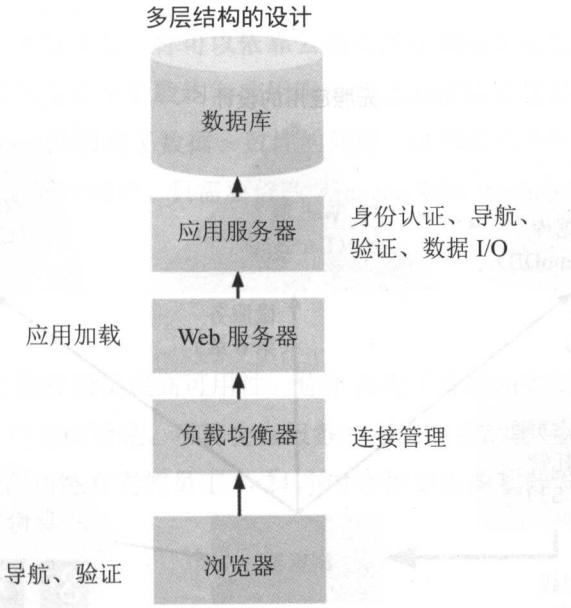
为了降低中间层成本并确保我们的应用能供上百万的用户使用，我们使用 Amazon Web Services（AWS）作为无服应用的后端。你将看到如何使用高可用、高可扩展、更便宜、更易维护的云服务来替换掉传统 Web 应用的服务器、数据库和负载均衡器。我们将讨论在开发此类应用时会碰到的一些安全性问题，并会介绍随着应用业务的扩展可能会用到的其他技术和工具。

我希望这本书能够让你看到新的可能性。以前非常费时费钱的应用开发或许能变成一个人在一两天内就可以完成的事情。随着技术进步和个人能力的提升，更多的梦想将会实现。一旦理解了这些技术的发展，你就会发现从前因为太难而几乎无法实现的目标，现在可以借由新途径来达成。读完本书，你将学会将创意变成真实应用所需的技能。

无服 Web 应用

在传统 Web 应用中，服务器是系统不可缺少的组成部分。尽管有时候服务器的前面还有负载均衡器或者专用 Web 服务器，但完成大部分工作的还是应用服务器。它完成一个应用所有的必要功能，包括存储用户数据、进行安全认证、控制流程等。应用的页面大部分仅仅只是为后端提供界面而已，尽管也会涉及一些控制导航的功能。使用这种许多人称之为多层架构的传统方式，系统一般会由浏览器、应用服务器和多个后端服务构成（见下图）。

使用无服的方式，可以移除所有这些层次架构，达到更直接的实现。与其仅仅把网页客户端当作应用服务器的界面展示，不如构建一个单页 Web 应用在浏览器中实现应用逻辑。这意味着你只需要一个简单的静态网页服务器，所有的交互都只不过是应用内容的传输而已，浏览器就像是一个应用容器。这样，最终的设计就是移除传统 Web 应用架构中所有的中间层次，允许浏览器直接连接到它所需要的服务上。

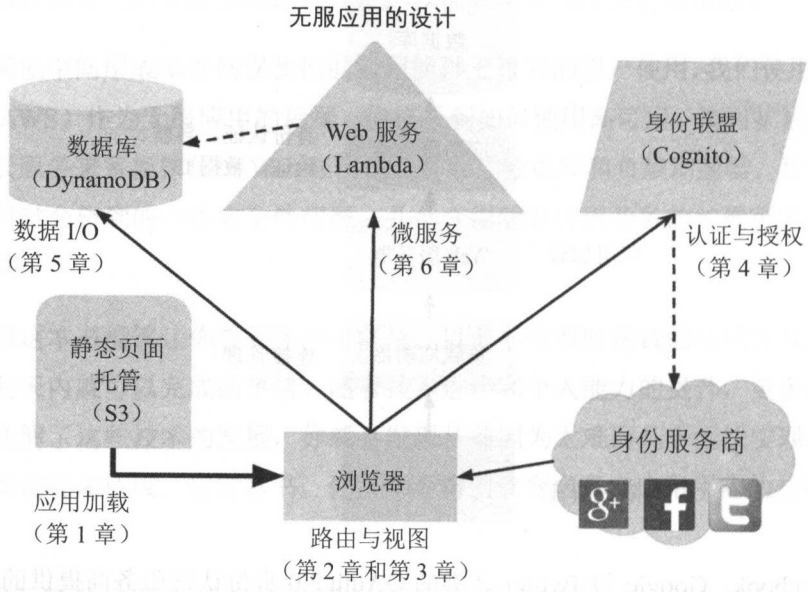


使用 Facebook、Google 和 Twitter 之类的 OAuth 2.0 身份认证服务商提供的服务，无须保存用户密码就可以创建用户身份。如果要存储数据，你可以在浏览器端直接使用 Amazon DynamoDB 之类的服务。在浏览器中无法执行的函数都可以使用 Amazon Lambda 微服务或者其他专门的 Web 服务来处理。除了能够简化架构，这种切换到 Web 服务作为后端的方式，还能让应用获得这些服务与生俱来的可用性和可扩展性优势。

你可能会好奇到底发生了什么，使这种方式成为可能。为什么现在在一个 Web 应用中，中间层的应用服务器变得可有可无呢？答案是，自从 2015 年以来，类似 Amazon 这样的云服务提供商开始对外提供服务的 API，这使得无服务器的方式成为可能，Amazon 本身也为如何使用他们的工具和基础设施提供了最好的示范。

基于 Web 标准搭建一个单页 Web 应用，而不是使用服务器端 Web 框架来完成，我们可以快速应用一些新兴技术。例如，我们不再需要将应用的数据模型绑定到任何一个对象层级或者数据同步机制上，因而能更方便地集成不同服务。既然我们所有的工作都倚赖于

Web，就不必拘泥于以前搭建 Web 应用的成见，可以用目前最新的技术来搭建应用（见下图）。



无服设计的好处

如果你在寻找一种快速搭建低成本 Web 应用的方法，无服 Web 应用很可能就是一个解决方案。不需要花费时间和精力了解传统 Web 应用技术栈的各个层级，采用这种方式你能更专注于实现业务功能，有人会为你操心运行维护和可扩展性的问题。接下来让我们深入探讨无服设计的好处，帮助你在考虑下一个项目中是否使用这种方式时做出更明智的决定。

零服务器

无服设计最明显的好处就是不需要维护服务器（不管是物理的还是虚拟的）。你不需要担心打安全补丁、监控 CPU 和内存使用情况、回滚日志、磁盘空间不足或者其他在维护自有服务器时经常碰到的运维问题。和大多数平台即服务（PaaS）方式一样，无服设计能让你专注于应用开发，而无须担心基础设施的问题。

易扩展

这种设计方式的另一大好处是，你可以依靠云服务供应商来扩展自己的应用。在做水平扩容时，不需要忙不颠地在几个负载均衡应用服务器之间保持数据的一致性，你可以直接连接 Web 服务，而它们已经解决了数据一致性的问题。这意味着不管你的应用有几个用户、几百个用户，还是几十万个用户，只需要修改 Amazon Web Services 控制台的一些设置就可以保证完美的运行。

高可用

另外，使用这种设计能轻松实现高可用性。你不必为了升级而关闭应用服务器，或者为了实现“热”部署而扩建基础设施。不再会有服务的重启或者部署包在服务器间的拷贝。最妙的是，Amazon 有一群训练有素的员工 7×24 小时守护着你的基础设施，一旦发现问题随时能够响应。

低成本

这些服务的成本可以非常低。使用无服的方式以及利用 Amazon 的免费套餐（Free Tier），一个月支付几美分就可以运行你的应用。一旦超过了免费额度，其费用经常也是随着你的用户量线性增长的（考虑费用最高的情况）。我们在这本书里构建的应用就算扩展到 100 万的用户，一天也只需要花费一杯咖啡的钱。

（微）服务友好

这种方式可以轻松适应微服务或者其他的面向服务架构。你可以在系统中引入特定的服务以实现自定义身份认证、验证或者异步数据处理。如果有必要，你甚至可以重新引入应用服务器，渐进式地重构应用。反之，如果一开始就使用一个中间层来控制所有的安全证书，就很难切换到需要认证的 Web 服务上。这些应用服务器没办法像无服应用一样，在应用层管理身份信息。

代码更少

在传统 Web 应用里，一些操作（比如导航）在 Web 客户端和服务器端都需要执行，造成了代码的重复。有时候，这种重复工作并不明显，尤其当服务器代码是用不同的语言写时。而在无服应用中，应用逻辑都移到了客户端，很容易保证应用内不再有重复的代码。将应用逻辑代码放在一个位置（以及用一种语言实现）帮助我们解决了这个问题。

此外，无服的方式更便于构建和排错，因为系统的组成部分变得更少了。Web 应用天生就是分布式的，也就是说，正如 CAP 理论所述[1]，它们在同一个网络的节点间传递消息（一般是以请求和响应的形式），限制它们的是实现方式。

有些应用会比其他应用更分散（more distributed）。一个系统越分散，就越难排错。移除应用中的中间层能减少其分散的程度。在我们这个简单的应用中，如果一个客户端需要从一个数据库中获取数据，就会直接连接数据库，而不是通过中间层连接。这就意味着系统中的网络节点更少，也意味着如果出现问题，需要定位的地方更少。

如上所述，构建一个无服应用的理由有很多。学完本书，你就会明白为什么这种方式如此强大。了解了无服应用的这些优点，我们再来看看它有哪些限制。

无服设计的限制

尽管无服架构有许多优点，但它也不是适用于所有类型的应用。为了享受这种设计带来的益处，你必须接受一系列的限制。如果你的应用不能适应这些限制，那么它很可能不是最合适的构建方式。所以在搭建应用之前，让我们一起看看这些限制。

供应商锁定

首先最大的限制就是你使用的 Web 服务必须支持第三方身份认证服务商，这样在云服务提供商的选择上就受到了限制。所以如果使用无服的方式，你就会依赖于第三方服务，供应商锁定也就成了一个问题。构建一个基于其他公司服务的系统，意味着这个应用的命

1 https://en.wikipedia.org/wiki/CAP_theorem

运和供应商公司的命运绑在了一起。如果供应商公司被收购、破产或者改变商业模式，你的应用不下大力气修改就很难在其他地方运行。所以，评估服务提供商的业务目标和长期稳定性与技术选型是同样重要的。

奇怪的日志

所有运维关注的事情，比如应用日志，在你使用无服设计之后会呈现新的形态。当你把所有请求都通过一台服务器路由时，记录下所有信息以查看用户正在做什么是非常简单的事情。没有了这种中心化设计，日志的记录必须由每个支撑应用的不同 Web 服务来实现。这些日志格式跟大部分应用服务器日志都不同，记录的数据也很可能是你不熟悉的。我们在后面第 8 章的“分析 S3 日志”会深入探讨 Web 服务日志的分析。

不一样的安全模型

对于无服应用，有些常见的安全隐患不复存在，但你将会遇到一些不熟悉的新问题。比如，为了安全而验证用户数据，结果不能在浏览器中安全地实现。你需要假设有些恶意用户可能会在浏览器中劫持证书而使用该证书授权的 Web 服务。使用无服的方式，意味着你不能把浏览器中的应用验证逻辑和安全验证逻辑放在一起，必须分开实现。

Amazon 提供的许多 Web 服务都能验证请求。你可以参考第 5 章的“数据访问和验证”一节内容利用 DynamoDB 来实现。然而，对于有些应用来说，很难只用 Web 服务提供的工具来实现充分的有效性约束。比如，在浏览器中直接编写文本时，你不可能放心地将写入的数据编码后存到数据库中，保证不会有跨站脚本攻击发生。因为攻击者不使用应用就能直接将这个数据添加到数据库。

这种情况下，你有（至少）两个选择。第一，可以假设某些用户可编辑的表可能包含未经验证的数据，然后针对性地设计系统的其他部分。比如，用户只能写入他们自己可读取的数据，这是可行的方式。第二，可以将某些写操作委托给自定义 Web 服务，比如可以使用 Lambda 函数来进行验证，并且以一种安全的方式写入数据。我们将会在第 6 章的“使用 Lambda 构建微服务”中详细介绍。

不一样的身份模型

外部身份管理是我们这本书构建的应用中的一个独特功能。使用 Web 服务来管理身份信息有很多好处，但对你来说这种机制可能有点陌生。与将用户信息和其他数据保存在一起的传统方式不同，这些用户资料会保存在一个独立访问的数据存储服务中。如果使用这种方式构建无服应用，一些在数据库中处理用户数据的方法（比如用一个 ID 关联一张 `User` 表）就没办法实现。

失去控制

此外，将所有请求路由到统一的中间层可以实现某种程度的控制，这在某些情况下是非常有用的。比如，拒绝访问攻击和其他一些攻击有时候可以在应用服务器上进行阻截。对你而言，放弃对身份认证的直接控制可能想一想都觉得可怕。我们后面在第 7 章会用一整章来专门探讨这些安全问题。

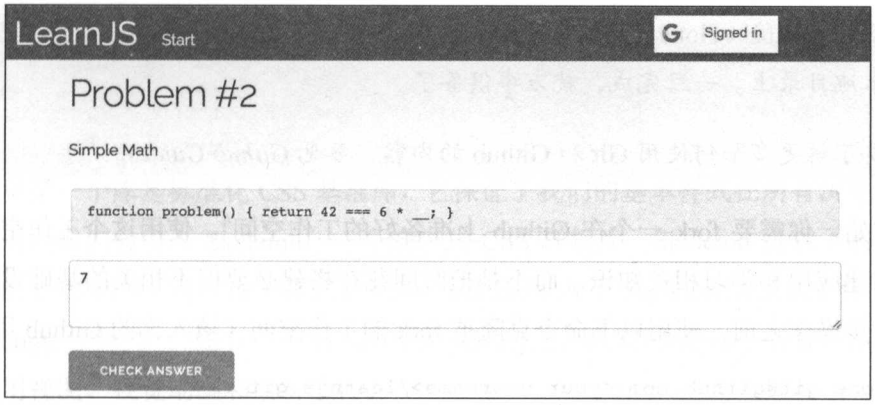
规模与成本的关系

最后，你需要了解这些服务的开销。虽然能够自动扩展应用这一点非常厉害，但易于扩展同时也意味着花钱更容易。你需要了解这些服务的定价策略以及当用户增加时这些价格的变化。我们后面会在第 8 章中深入讨论应用的成本。

既然你已经了解了无服 Web 应用的代价，我们可以开启这本教程，探索一下无服 Web 应用是如何实现的。在教程中，你可能会发现这种设计方式为你开发的 Web 应用带来的其他好处和限制。一旦知晓了无服应用的全貌，就可以判断下一个项目是否适合这种方式了。

使用自己的工作空间

为了学习无服 Web 应用的知识，我们将在本书中搭建一个 JavaScript 编程解题应用作为示例，名字叫 LearnJS。它会向用户展示一些简单的编程题，然后让他们用 JavaScript 作答，并按下按钮检查答案是否正确。这个应用的样子如下图所示。



我们将会从后往前搭建这个应用。本章我们将会部署它，然后测试，再加入一些应用逻辑。在那之后，我们再来思考架构设计。

如果你对现代开发者们拥护的这种迭代增量式开发风格不熟悉（我本来想把它叫作敏捷开发，但是它和敏捷的涵义还有所不同），这套流程看起来是完全错误的。还没有构建好应用我们怎么部署呢？还不知道要让应用做什么，怎么先写自动化测试呢？还有，我们是不是应该在动手之前先考虑一下架构设计呢？

如果你对此有疑问，没关系，我们将会一步步实现这个流程。一旦完成之后，你将会理解，甚至是赞同这种方式。不仅因为它是学习新技术的绝佳途径，而且也是构建软件的有效方法：前进一小步，评估当前状态，不断重复此过程，直到客户满意。

使用 Git 进行代码管理

Git 是一个代码管理系统。与其他系统一样，Git 能帮你跟踪代码的变动，保证代码安全，以及与他人共享。如果你对它不熟悉，可以看看下面的介绍。

Fork 一个项目意味着创建一个自有的副本。如果你在 Github.com 上创建一个账号，再查看你的工作空间项目[2]，应该可以在页面右上角看到一个“Fork”按钮。单击按钮，过几秒后，Github 就会为你创建这个项目的副本。现在你需要把它复制到本地电脑上，

2 https://github.com/benrady/learnjs

这个过程称为克隆（clone）。使用一个 Git 客户端，你能将整个工作空间都克隆到自己电脑的本地目录上。一旦完成，就万事俱备了。

想要了解更多如何使用 Git 和 Github 的内容，参见 *Github Guides*。[3]

一开始，你需要 fork 一个在 Github 上准备好的工作空间[4]。使用这个工作空间能让你专注于构建应用和学习相关知识，而不是把时间花在搭建必要但不相关的基础设施上。在进行下一步操作之前，使用以下命令克隆被 fork 的工作空间（填入你的 Github 用户名）：

```
$ git clone git@github.com:<your username>/learnjs.git
```

要理解接下来会做什么，首先需要理解现有的代码。刚才克隆的工作空间中有一个 `public` 文件夹，里面包含了一个空应用。这个应用没有任何功能，它的标记（markup）也很快就会被替换，但它包含了所有我们需要的基本工具。

在构建应用时，我们主要会修改 `public` 文件夹下的三个文件：`index.html`、`app.js` 和 `app_spec.js`。我们会在 `index.html` 中添加应用的标记。这就是我们单页 Web 应用中的“单页”。使用自己顺手的文本编辑器，打开这个文件，查看一下内容。在<head>元素中，你将会看到这个预备好的工作空间里我们必须使用的一些库。

```
<head>
  <meta charset='utf-8'>
  <title>Learn JS!</title>
  <link rel='stylesheet'
    type='text/css'
    href='//fonts.googleapis.com/css?family=Raleway:400,300,600'>
  <link rel='stylesheet'
    href='https://cdnjs.cloudflare.com/ajax/libs/normalize/3.0.2/normalize.min.css'>
<link rel='stylesheet'
    href='https://cdnjs.cloudflare.com/ajax/libs/skeleton/2.0.4/skeleton.min.css'>
  <script src='https://code.jquery.com/jquery-2.1.4.min.js'></script>
  <script src='/vendor.js'></script>
  <script src='/app.js'></script>
```

3 https://guides.github.com

4 https://github.com/benrady/learnjs

```
  <style type="text/css" media="all">
    body { margin-top: 30px; }
  </style>
</head>
```

应用里第一个库是标准化 CSS 基准库，它保证了我们的基本样式在所有浏览器中都是一致的。后面我们会引入 Skeleton，一个响应式 CSS 样板库。Skeleton 提供了响应式网格和一些小的 CSS 组件，可以用于样式和布局。我们还引入了 Skeleton 的默认字体 Raleway，托管在 Google 字体库 。还有一个引入的库是 jQuery 2。我们用 jQuery 来做许多事情，比如搭建应用视图、视觉动画和监听事件。

第二个文件 `vendor.js` 包含的库要么是我们应用自定义的，要么是不够流行而没有放在 CDN 上的类。迄今为止，这个文件包含的唯一东西是一个 AWS[5]（版本号为 2.2.4）JavaScript 开发库的自定义子集，包含了以下几个服务类库：

- CognitoIdentity
- CognitoSync
- DynamoDB
- Lambda
- STS

`<head>`元素里的另一个脚本为 `app.js`。到目前为止，这个文件是空的。这里就是我们添加用 JavaScript 编写的应用逻辑的地方。它不仅会包含应用的领域特定逻辑，而且包含像路由器、模板函数和数据绑定代码这样的基础设施。

`<head>`元素里的最后一项是一个`<style>`元素，里面目前放置着我们应用的 CSS 规则。当然，在某个时间点你可能会想把这些规则抽出来放到它们自己的文件中，但目前让它们内嵌在其中会更简单。

应用的`<body>`元素目前只有一个简单的头部和一些文本。我们很快就会把它替换成一个标题页面。等到我们学完这本教程，`<body>`元素将会包含应用的所有标记。这个标记将

5 https://sdk.amazonaws.com/builder/js/

定义我们的用户界面和应用视图。

learnjs/1000/public/index.html

```
<body>
  <div class='container'>
    <h1>It works!</h1>
    <div>
      <span>You're ready to start!</span>
      <span>Skeleton 2, jQuery 2, and AWS libraries are included.</span>
    </div>
  </div>
</body>
```

工作空间内还包括一个名叫 Jasmine 的测试框架。空间内的 public/tests 文件夹包含了一个测试执行器和一个空的测试组件（public/tests/app_spec.js）。当我们编写测试用例来保证应用的功能时，就把它们添加到这里。

本地执行

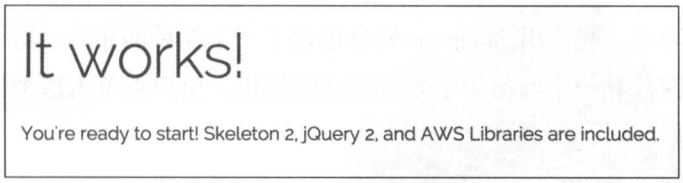
既然了解了目前的进度，就把应用启动起来，然后把自己设想成用户来看一看。接着我们可以对其进行评估，做一些小修改，部署我们的第一个版本。要做这些事，需要一台在本地运行的 Web 服务器来为应用提供服务。

预先准备好的工作空间提供了一个叫作 sspa 的包装器脚本。对这个脚本的使用将贯穿全书，它将实现包括配置 AWS 服务、构建代码包和部署应用等简单任务。这个脚本代码非常易读，如果想知道它是怎么工作的，尽管打开看一看。为了在本地启动一个包含 public 文件夹下的内容的开发服务器，在预设的工作空间主目录下执行以下命令：

```
learnjs $ ./sspa server
```

sspa 脚本的 server 指令会启动一个简单的 Python Web 服务器提供静态内容。当然，你也可以使用自己的 Web 服务器。只需让预备的工作空间来提供 public 文件夹下的内容，一切搞定。我们在书中所做的事情都不会涉及特定的 Web 应用开发工具。不管你喜欢使用什么工具，都可以在你最舒服的工作环境中应用本书中的所有技术。

一旦搭建好 Web 服务器并运行起来，看一下我们的应用。如果你是用预备工作空间中自带的服务器，可以打开习惯的浏览器，访问 http://localhost:9292。你应该可以看到以下内容：



这就是我们的应用。目前还没有什么内容。尽管可以试着为应用规划一长串的功能，但因为还不知道究竟我们想要些什么，超前规划似乎会变成徒劳。相反，我们会给应用添加一个标题页面，然后决定接下来做什么。

创建着陆页

当用户第一次加载我们的应用时，我们想展示给他们一个页面，既能快速解释我们的应用是什么又能提供清晰的入门指南。这个页面通常被叫作着陆页（landing page），我们的应用中也会添加一个这样的页面。

LiveReload 和 LivePage

不管你使用哪种 Web 服务器，我都非常推荐在实际项目中使用某种自动刷新页面的工具。有很多现成的选择。有一个与我们的预备工作空间配合使用的好工具是 Google Chrome[6]中的 LivePage 插件。它安装和使用起来很容易，只要你修改了代码，它都会运行测试脚本并重新加载应用。这样的工具建立了一个快速反馈闭环，你可以用来检验样式和布局变动或者执行一些客户端测试。

另外一个选择是名为 LiveReload 的工具。LiveReload 既可以用作一个单独的 Web 服务器，也可以在开发中用作自动重载 Web 应用的协议。无论什么时候服务器检测到硬盘

6 https://chrome.google.com/webstore/detail/livepage/pilnojpmdoofaelbinaeodfpjheijkbh?hl=en

文件的改动，都会及时通知客户端的 JavaScript 库（或者浏览器插件）。LiveReload 有多种形式，有 LiverReload 应用[7]和像 LiveReloadX[8]这样的命令行工具。如果你希望在 Node.js 中运行，live-server[9]或者 grunt-livereload[10]可能更适合你的需求。

为了构建着陆页，我们用 Skeleton 网格创建了一个简单的布局。这个页面会包含一些标题文本、一张图片和一个经常用于号召行动的按钮。用户将单击这个按钮来启动我们的应用。

如果你之前不熟悉 Skeleton，现在是一个熟悉它的好时机[11]。当然，它里面其实并没有很多东西（所以才叫这个名字）。它的文档非常规范，而且有许多案例，所以很容易学习。花 5 分钟时间过一遍，就能理解它能做什么以及是怎么实现的。

为了展示这个网格，我们将厚着脸皮从 Skeleton 着陆页的案例中“偷”些代码。使用你的文本编辑器，打开 `index.html`，将这些标记添加到应用页的`<body>`中：

learnjs/1100/public/index.html

```
<body>
  <div class='container'>
    <div class='row'>
      <div class='one-half column'>
        <h3>Learn JavaScript, one puzzle at a time.</h3>
        <a href='' class='button button-primary'>Start Now!</a>
      </div>
      <div class='one-half column'>
        <img src='/images/HeroImage.jpg'/>
      </div>
    </div>
  </div>
</body>
```

7 https://github.com/livereload/LiveReload

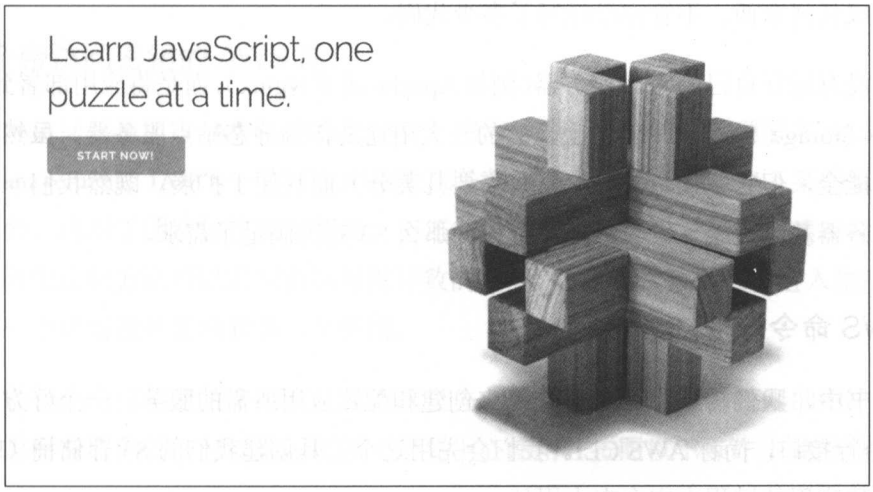
8 http://nitoyon.github.io/livereloadx/

9 https://github.com/tapio/live-server

10 https://www.npmjs.com/package/grunt-livereload

11 http://getskeleton.com/

在你的浏览器中，应该能看到我们的着陆页，而不是之前的文本了（见下图）。单击按钮并不会触发任何操作，但这并不是错误。



既然我们已经添加了一个着陆页，就有东西可以去部署了。虽然不是很丰富，但也足够用来走部署流程，足够我们在为应用添加更多功能之前把一切理顺。

部署到 Amazon S3

当启动新项目时，经常会碰到很多未知的风险。那些问题可能根本不在预料之中，会让你浪费很多时间。如果能识别并避免这些风险，可以少一些挫败感、头疼和痛苦。

有一个很容易避免的风险就是部署问题。我们不想等开发完成之后再部署。我们关于应用如何运行的假设有可能是错误的。如果基于这些错误的假设构建应用，风险和潜在痛点会增加。因为我们是在个人电脑上开发应用然后部署在服务器上的，Web 应用经常面临一堆问题。这两种环境很可能完全不一样，等你理解了它们所有的不同点，就会发现自己是在白费力气。

通过提前部署，我们就知道自己的部署流程是否做了正确的配置。我们可以检测生产

环境是否可行，确保没有任何权限之类的问题。最重要的是，我们解决了所有必要的任务，证明我们可以把应用从头到尾完成。在你的应用交付给用户使用之前，从他们的角度，你就是没完成任何东西，不管你已经写了多少代码。

我们没有运行自己的Web服务器（例如Apache或者Nginx），而是将应用部署到Amazon的Simple Storage Service（S3）上。它的一大用途是作为静态站点服务器。虽然不如其他服务器功能全，但它很便宜（一个月只需要几美分）而且便于扩展。既然我们一直在尝试避免在服务器基础设施上投入时间和金钱，那么S3完美满足了需求。

搭建AWS命令行接口

在本书中，我们需要和AWS交互来创建和配置应用所需的服务。一个好方法是使用AWS命令行接口，简称AWS CLI。我们会先用这个工具创建我们的S3存储桶（bucket），在后面构建应用的过程中它会有大用处。

如果你还没安装AWS CLI，那就赶快安装，并且配置好管理员权限。Amazon推荐使用`pip`，一个Python包管理器，来安装AWS CLI。如果没有`pip`，也可以使用`easy_install`，它也是一个Python包管理器。你需要使用Python 2.7以上版本。根据你安装的管理器，执行以下对应的命令：

```
$ sudo easy_install pip
$ sudo pip install awscli
```

在Debian/Ubuntu系统上，你可以用`apt`来安装`pip`：

```
$ sudo apt-get install python-pip
```

在OS X 10.11上安装AWS CLI

如果在OS X 10.11（El Capitan）上用`pip`安装AWS CLI出现问题，你可以在`install`命令[12]后面添加`--upgrade`和`--ignore-installed six`参数。

12 https://github.com/aws/aws-cli/issues/1522#issuecomment-159007931

为了使用这个工具，你需要配置它。首先，要在 AWS 账号上创建一个带管理员权限的用户。

创建一个带访问密钥的 AWS 用户

授权你访问其他所有 AWS 的那个 AWS 称为身份认证和访问管理服务（Identity and Access Management，IAM）。你可以用这个服务创建一个能访问你账户下某些服务的独立用户。显然，这对于团队协作很有帮助，但还能用它根据角色或者任务创建不同用户。类似这样的分组访问方法可以限制密钥泄露导致的漏洞，防止测试数据直接进入生产数据库，或者防止一个应用意外影响到另一个应用。

你可能想为应用创建很多用户，我们这里先创建第一个带管理员权限的用户。该用户有你账户里所有服务的访问权限，所以一定要小心使用。按照以下操作步骤创建用户：

1. 打开 AWS 控制台[13]，如果有必要，注册一个账号。
2. 进入“Security & Identity”下面的“Identity & Access Management”服务。
3. 在左边侧边栏，点击“Users”。
4. 点击“Create New Users”创建一个新用户。我们将会使用这个用户账号来部署应用。
5. 为用户挑选一个名字（比如 learnjs），填在第一行。
6. 勾选上“Generate an access key for each user”（为每个用户分配访问密钥）复选框，点击“Create”。
7. 按照提示下载证书。

证书由两个密钥组成：访问密钥（access key）和私钥（secret key）。这两个密钥都在

13 http://console.aws.amazon.com

你下载的 CSV 文件里，也可以直接在网站上看到。Amazon 只给你一次操作机会，所以马上下载下来，否则你只能重新创建密钥了。

不需要为这个用户提供密码。

一旦拿到这些密钥，就可以完成 AWS CLI 的设置。用管理员用户运行 `aws configure` 命令，然后按照提示输入密钥信息。如果它要求你设置默认区域，输入 `us-east-1`。

```
$ aws configure --profile admin
AWS Access Key ID [None]: JFAKEKEYSRRETDMAAKIA
AWS Secret Access Key [None]: 2Jdw+ThI5iSafAKeKeY4ExamPLEsHAONXn32Af/sm
Default region name [None]: us-east-1
Default output format [None]:
```

Joe 问：

如果我想用其他区域呢？

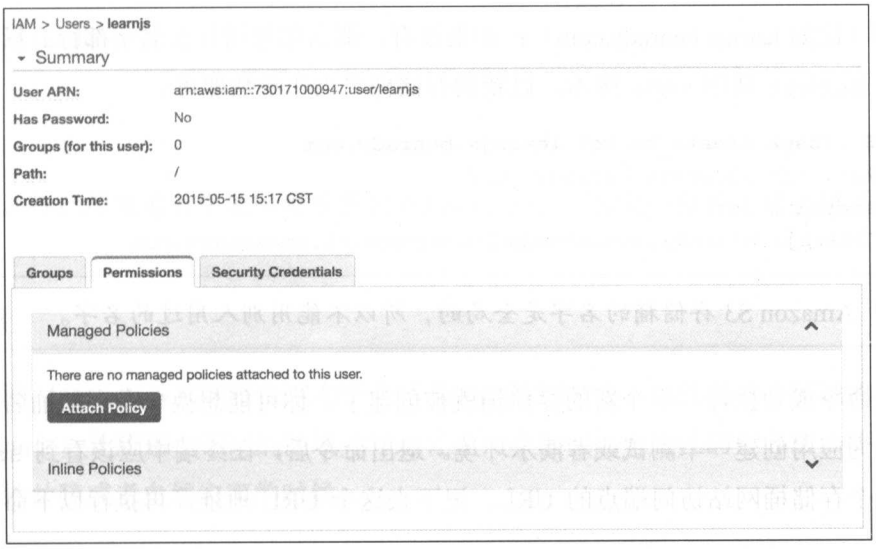
如果你对 AWS 比较熟悉，很可能在一个非 `us-east-1` 的区域有几个已经配置好的服务了。当然你也可以在管理员界面中修改区域配置，但注意并不是所有的服务都适用于所有的区域。另外，在其他区域 `sspa` 脚本可能没法正确处理 URL 请求和其他资源名。

如果你用了非 `us-east-1` 的其他区域，需要验证一下你所使用的服务在那个区域是否可用。另外，你可能需要修改下 `sspa` 脚本以适配你的区域名。

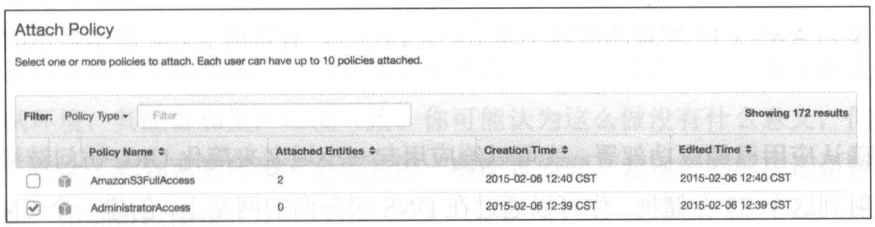
既然 AWS CLI 已经配置好了，在主目录下应该能发现一个新文件 `~/.aws/credentials`。这个文件的内容如下所示（当然密钥是不一样的）：

```
[admin]
aws_access_key_id = JFAKEKEYSRRETDMAAKIA
aws_secret_access_key = 2Jdw+ThI5iSafAKeKeY4ExamPLEsHAONXn32Af/sm
```

在创建用户并且保存证书之后，你需要通过创建访问策略来授予该用户访问的权限。回到 AWS 控制台的用户列表页，你应该可以看到最近创建的用户。点击该用户，进入用户概要页。它应该看起来如下图所示。



接下来，你需要创建一个策略，规定该用户能访问哪些服务。找到用户概要页的“Permission”模块，点击添加新托管策略的链接。你将会看到一系列如下图所示的策略信息。



勾选上“AdministratorAccess”旁边的复选框，然后单击“Attach Policy”按钮，就大功告成了。现在你可以创建一个 S3 存储桶（bucket）。然后，就可以部署应用了！

创建一个 S3 存储桶

基于已有的管理员用户，你可以创建一个 S3 存储桶。预备工作空间中的 `sspa` 脚本包装了一些 AWS CLI 的命令，不仅能够创建存储桶，而且能够把它配置成服务于我们应用的静态文件 Web 服务器。如果你还有一个想指向这个应用的域名，这里必须将它输入成存储

桶的名字（比如 learnjs.benrady.com）；如果没有，那么你想用什么名字都行。只需要输入 `create_bucket` 调用 `sspa` 脚本，以新的存储桶名当作参数即可。

```
learnjs $ ./sspa create_bucket learnjs.benrady.com
make_bucket: s3://learnjs.benrady.com/
Website endpoint is: \
  http://learnjs.benrady.com.s3-website-us-east-1.amazonaws.com
```

Amazon S3 存储桶的名字是全局的，所以不能用别人用过的名字。

如果命令成功执行，一个新的存储桶就被创建了。你可能想换一个存储桶名再运行一次命令，为应用创建一个测试或者演示环境。退出命令后，在终端中应该看到 `sspa` 脚本返回了一个存储桶网站访问端点的 URL。记下来这个 URL 地址，再执行以下命令部署应用：

```
$ ./sspa deploy_bucket learnjs.benrady.com
```

好了！现在把 `sspa` 脚本刚才显示的网站访问端点输入到浏览器中，就可以看到我们的应用了。它上线了！如果没看到，可以仔细检查 AWS 控制台中 Amazon S3 的网站端点地址。如果为 AWS CLI 配置的区域不是 `us-east-1`，有可能 `sspa` 脚本显示的地址是不对的。

一旦确认应用已经成功部署，就可以给应用起一个域名来简化 URL 访问链接。你需要把域名映射到这个 S3 存储桶。你可以通过在 DNS 服务商的网站上，创建一个 CNAME 项，把端点 URL 当作记录的值，来实现映射。更多有关如何实现的详细信息，参见附录 B 的内容。一旦做完这些，部署工作就全部完成了，我们就可以继续下一步的工作。

首次部署

你已经把应用成功部署到生产环境中了。有了这一次的经验，后面再修改应用，你应该有信心完成部署了。这种方式将帮助你小步前进，专注于怎么让应用变得更好以及怎么

把它推向潜在用户。

下一步

至此，你已经知道怎么把应用部署到 Amazon S3。下面是一些你可能会感兴趣的延伸话题。

AWS 区域

所有的 AWS 服务都运行在遍布于全球各地的数据中心，也就是前文所说的“区域”（region）。让应用在不同的区域并行运行是一个确保高可用性的好办法，即使在面对灾难性故障或者自然灾害的时候。

在选择区域的时候，首先保证你需要的服务在那个区域是可用的。另外还必须考虑你的用户在什么区域，其他非 AWS 的支撑服务器在什么区域（比如邮件服务器）等等。

创建测试环境

我们在本章中刚接触它，但是随着本书的内容一点点深入，后面会搭建一个完整的测试环境，其配置和生产环境一致。你可能认为这么做没有什么意义，但走完整个过程，你会了解到两套环境下会面临的一些配置问题。是现在就解决还是暂时搁置这些问题，由你自己决定。

IAM 锁定

我们在这个应用中创建的管理员用户可以访问账号里的所有服务。等你对需要的服务更熟悉之后，更好的方式将是移除这个用户的所有管理员策略，添加一些细粒度策略，只允许它访问真正需要的服务。

至此，我们已经为接下来的工作做好了准备。在下一章中，我们将会添加一个客户端路由到应用中，支持不同的应用视图，以及解决一些按钮不管用的问题！

第 2 章

基于hash事件的视图路由

在第 1 章中，我们部署了 Web 应用的第一个版本，但它基本上是一个静态站点。单击按钮没有任何响应，并且只有一个页面。我们需要给应用添加更多的页面，并且使这些页面动态化，即能够响应用户的操作并且提供交互式体验。

在静态站点中，每个页面的内容是固定的，通过超链接导航到其他页面。传统的 MVC Web 框架通过服务端路由器将 URL 和控制器关联起来，改善了这一点：控制器使用视图和模型生成动态内容。单页应用将路由器逻辑移动到浏览器端，来实现甚至增强这个功能，避免了请求往返于服务器。这使得客户端路由器成为所有单页应用必不可少的组件。通过创建路由器，我们可以在应用中实现只加载单个 HTML 页面就能够支持多视图。

你可能很好奇，路由器是如何避免向服务器发送请求的？毕竟，万维网是由超链接文档组成的。如何做到从一个页面移动到另一个页面而无须下载新的页面文件呢？得益于浏览器的常驻策略（long-standing policy），当用户点击了一个仅仅包含 hash 标签（如 #home）的超链接时，这个标签会被添加到 URL，但是该页面不会被刷新。这个标签被称作 URL hash。

使用 JavaScript，我们能监听当这个 hash 变化时浏览器触发的事件。我们可以利用浏览器的这个行为来为单页应用构建路由器。当 URL hash 发生变化时，路由器会查找一个函

数，该函数生成我们想要展示给用户的标记（markup）。这使用户视图改变时应用不会被重新加载；同时维持相似的 URL 含义，用户可以保存、收藏、复制和分享它。

在本章中，我们将构建一个路由器，该路由器利用 hash 事件触发动态标记的生成与显示。它使用的是预定义的路由（将 URL hash 和视图函数关联起来）。这些视图函数将为视图创建标记和行为。这样，我们不仅可以在应用中添加路由来创建新视图，同时也有了一种便捷的导航方式。一旦路由器开始工作，就能通过 jQuery 监听事件来加载应用。本章末尾，我们将成功地把我们的站点变成一个真正的单页应用。

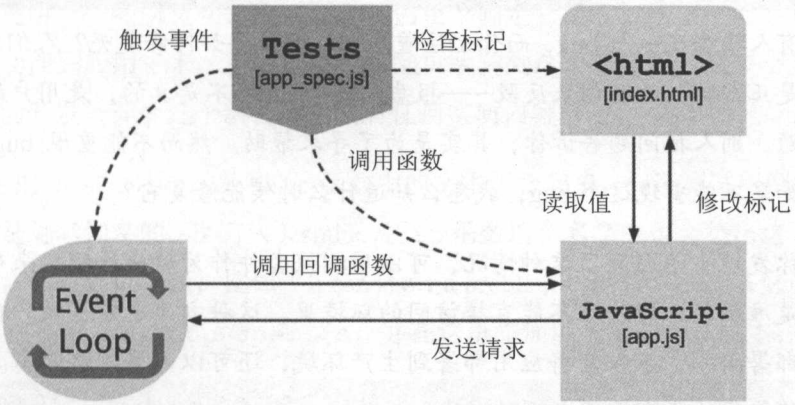
设计可测试的路由器

虽然有一些例外，但是要在你的本地环境中运行 AWS 服务通常是不可能的。让应用的后台在 AWS 后端运行，你可能会担心这样难以测试。如果要创建一个快速的反馈环路，以便在开发的时候确认程序运行是否正常，就需要改变以往构建应用时就做测试，而在开发环境中运行全部服务栈的传统方法。相反，我们采用测试优先（test first）的方法编写单元测试，通过单元测试来测试路由器。先编写测试用例，再设计路由器，这将确保设计出来的路由器易于测试。虽然只在本章着重讲测试，你完全可以把这些技术运用到本书的其他章节。

路由器的设计对应用的整体可测试性有极大的影响，这也是我们从测试开始介绍的原因。我们想要创建一系列能快速检查应用是否正常的自动化测试，不依赖后端服务来执行测试用例。为了构建这个路由器，我们将通过测试循序渐进地实现它的功能。我们将逐一编写这些测试，与应用的开发同步进行，以确保所有的代码是可测试的。我们会避免测试完整的工作流，而专注测试小粒度行为，所有的测试几毫秒就能运行结束。在测试中可以忽略竞争条件和其他时序问题，因为它们不需要向服务器发送异步请求。

一种开发可测试应用的方法是，在设计中引入“线缝”（seam）。换句话说，“线缝”就是明确的界限，在这些边界上测试用例能轻松地调用行为、检查输出、模拟交互。在我

们的应用中线缝就是标记。如果测试用例能获取该标记，就能检查它。将 JavaScript 封装，能生成另一种线缝。测试用例能调用 JavaScript 意味着它可以触发 JavaScript 来验证行为。浏览器也可以扮演一个线缝，让我们的测试用例通过触发事件来模拟用户操作。在下面这张图中可以看到一些测试线缝。



一个易于测试的应用，天然有助于解耦[1]，而解耦则能改善整体设计。当我们创建路由器时，也会使用这些测试用例，通过重构来优化代码的设计。我们会一点一点地重构，确保只添加非要不可的代码，尽量保持代码简洁。最后得到的应该是一套可靠的测试用例，能够支撑一个简洁、实用并且易于测试的应用。

运行 Jasmine 测试

我们将使用 Jasmine 测试框架来编写测试用例。Jasmine 与 RSpec 和大多数 xUnit 框架类似。如果你曾经用过这些框架，就会觉得得心应手。如果没用过，可以跟着本书一步步来，也可以先查看 Jasmine 的文档[2]。一旦掌握了规律，这件事就会变得很简单。

我预备的开发环境里有一个测试执行器。如果你已经打开了应用，将/tests/index.html

1 https://en.wikipedia.org/wiki/Coupling_(computer_programming)

2 http://jasmine.github.io

追加到地址栏中的 URL 之后，应该会看到测试结果。现在，它应该会说我们还没有测试用例。和应用的其他部分一样，如果使用 LivePage 插件或者 LiveReload 服务器，当你修改了应用或者测试用例时，页面将自动刷新。

在生产环境中运行测试

如果有人报告了一个 bug，而你无法重现它，那么要如何修复呢？人们总是用“在我的机器上是正常的”的托词来反驳——报告的这个 bug 不是真的，是用户的问题而不是程序的问题。别人把问题告诉你，其实是为了寻求帮助。然而不能重现 bug 让人寸步难行，毕竟如果不能重现这个 bug，我怎么知道什么时候能修复它？

如果你发现自己遇到了这种情况，可以用测试套件作为健康检测，来确认你关于应用的判断是准确的，即使在不能直接访问的环境里，这种方法也能奏效。在预备好的工作环境里部署脚本，不仅能将应用部署到生产环境，还可以部署测试脚本。这意味着你可以在运行应用的所有设备上运行测试。

如果有用户发现问题，一个快速的解决方案是要求用户浏览/test，确认一切是否正常。如果不是，则请用户复制和粘贴输出内容，并发送给你。然后你就可以尝试重现失败的测试，而不用重现用户报告的错误了。

Jasmine 将测试脚本封装在回调函数中，把这些测试组织到一起，传递给回调函数两个函数作为参数：describe 和 it。it 函数用于编写单个测试，describe 函数允许我们添加这些测试的上下文和设置。我们将添加一个外部 describe 为应用保存所有测试用例，然后添加一个 it 函数来执行我们想要的测试用例。

编写第一个测试用例

在编写测试用例之前，需要确定我们想要测试的功能。用简明的文字列出这些功能，并且用一致名字为测试命名，当我们稍后回过来看的时候就知其所以然了。测试用例的名字应该体现为什么它的代码做了这件事，而不是表明它是怎么做的。

我们需要路由器的原因，是要在应用中支持多视图。现在，我们只有一个着陆页（landing page）。为了实现路由器的功能，我们将编写一个测试用例来确认我们还能再创建一个视图。这不仅仅能帮助我们实现想要的功能，而且清晰地说明为什么需要它。

我们要创建的第二个视图将显示应用内的编程问题，所以称为题目视图（problem view）。这将是用户与应用交互的主视图。我们还不确定它是什么样子，但是我们起码能添加足够多的功能到应用中来，使我们能从着陆页跳转到这个新视图。这个小步骤能帮助推进整个应用的开发，即使这个新视图中没有任何实质内容。

Jasmine 会把 `describe` 函数和 `it` 函数中的文本组合起来，构造测试用例的全称。要保证这些名字是见名知义的。我们从 Jasmine 中 `it` 函数这个名字中也会猜出这个测试用例是如何命名的，你应该读得懂，比如“It can show a problem view”。在准备好的开发环境中，打开 `public/tests/app_spec.js`，并在这里添加以下新的测试脚本。

learnjs/2000/public/tests/app_spec.js

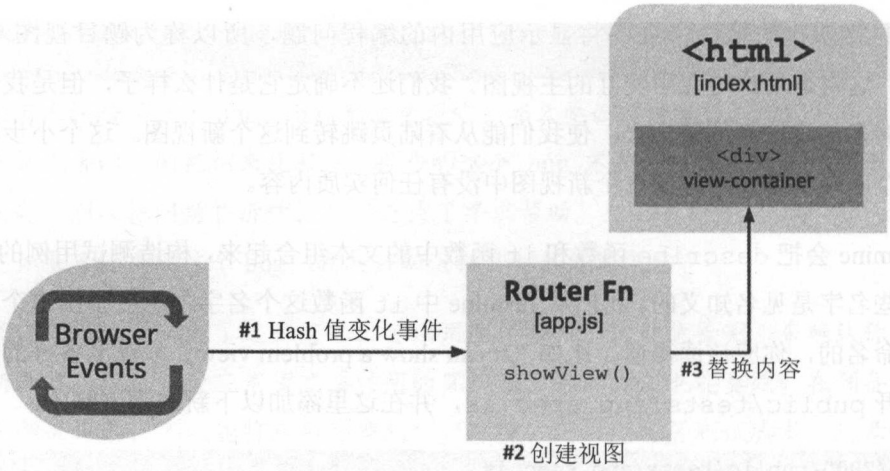
```
describe('LearnJS', function() {
  it('can show a problem view', function() {
  });
});
```

既然已经为测试用例起好名字，现在可以开始写测试脚本了。在这个测试脚本中，将测试调用一个 JavaScript 函数，我们称之为路由函数（router function）。路由函数的作用是，根据 URL hash 所定义的路由找出并显示合适的视图。我们将这个函数命名为 `showView`，并将它添加到名为 `learnjs` 的命名空间里[3]。这个函数的调用结果是，创建该视图的标记，并且把它添加到应用中。它将 URL hash 作为一个参数，来确定要加载的视图。在这个测试中，我们将传入在应用中代表第一个题目视图路由的 hash 值：`#problem-1`。

`showView` 函数现在还不存在，但这没有关系。我们将用这个测试用例驱动 `showView` 函数的设计。下面的图显示了构建路由器时的第一个传递过程。需要注意的是，在我们的应用中还没有办法调用路由函数。最后，当浏览器触发 `hashchange` 事件时，我们应用的

3 http://eloquentjavascript.net/10_modules.html#h_NitCO6r9Hn

另一部分将调用 showView()。它将通过文档定位 API[4]，window.location.hash，获取并传入当前 hash 值。当 showView 被调用后，我们将创建视图，并将其插入到页面中。



在测试脚本中我们要确认的是，路由器已经将题目视图标记添加到一个视图容器中（view container）。视图容器是路由器另一个必不可少的部分。它是用来承载视图标记的页面元素。当路由器添加一个新视图时，该视图容器中的所有标记将被替换。

为了写测试脚本，我们将使用 jQuery 来选择页面元素[5]。我们声明在应用中有一个拥有 view-container 样式类的元素，并且在这个元素内部还有一个元素，该元素拥有 problem-view 类。下面通过 jQuery 的元素选择器来实现上述内容，并且声明所选择的元素数量等于 1。

learnjs/2100/public/tests/app_spec.js

```
describe('LearnJS', function() {
  it('can show a problem view', function() {
    learnjs.showView('#problem-1');
```

4 https://developer.mozilla.org/en-US/docs/Web/API/Window/location

5 https://learn.jquery.com/using-jquery-core/selecting-elements/

```
    expect($('.view-container .problem-view').length).toEqual(1);
  });
});
```

如果你的 LivePage 或者 LiveReload 工作正常，当你保存这个 spec 文件时，测试执行器会重载并执行这些测试，并告知你该测试失败：

```
1 spec, 1 failure
LearnJS can show a problem view
ReferenceError: learnjs is not defined
```

这些测试如我们所预期的那样失败了。因为我们还没有为应用添加任何行为，也没有创建应用运行的命名空间。要使这个测试通过，第一步就是要做这些事。

路由函数

既然有了一个失败的测试，我们可以写路由函数了。为了让路由器通过测试，必须做三件事。我们需要在标记中创建一个视图容器，以便路由器有位置来插入视图。我们需要编写 `showView` 函数，并实现一个简易版题目视图。最后，我们需要创建一个命名空间，路由函数和应用的其他部分能在其中运行。完成这三件事，路由器的第一个版本就完成了。它只能支持从一个视图跳转到另一个视图，但这是一个好的开始。

我们的测试用例表示无法找到 `learnjs` 命名空间，那么我们就来解决这个问题。

创建命名空间

你可能很好奇为什么需要一个命名空间。虽然我们可以自行定义 `showView` 函数，但开发应用时必须尽可能避免使用已定义在全局作用域下的函数和变量，否则会产生命名冲突（name collision）而出现问题。把我们的应用放在单独的命名空间里可以避免命名冲突，确保它在当前和未来的浏览器中都能运行。

主流浏览器提供的 JavaScript 运行时环境包含了一系列不同的 API 接口。为了使用这

些接口，浏览器定义一个 window 对象作为顶层入口。你可以引用这个对象来调用接口。举个例子，你能通过调用 window.location.toString() 获取当前 URL。你也可以通过给 window.location 赋一个新的 URL 字符串值来用程序在浏览器中实现导航。

除了是浏览器接口的访问入口，window 对象还是全局变量的容器。调用浏览器接口不需要带 window 前缀，因为它们是全局的。例如，你可以使用 location.toString() 来代替 window.location.toString()。不幸的是，这也意味着你所创建的全局变量名有可能和接口冲突。即使你能保证现在你的变量名没有冲突，但是很难保证与浏览器制造厂商未来引入的 API 也不冲突。

幸运的是，有个简单的方法能避免这个问题。我们可以为应用创建一个命名空间。命名空间实际上是一个承载应用程序的 JavaScript 对象。我们最先添加到 app.js 文件中的内容就是命名空间。

learnjs/2100/public/app.js

```
'use strict';
var learnjs = {};
```

看！多简单。这项简单而强大的技术为我们的应用提供了某种结构，有助于避免命名冲突。我们可以放心地将应用的函数和变量添加到这个对象中，而不用担心它们是否与 window 全局对象或者其他类库定义的全局变量冲突。

我们同时引入了'use strict';，这条语句会通知浏览器，在执行我们的 JavaScript 代码的时候，要启用严格检查模式。这就是说，我们不会忽略问题，而是会得到一个很棒的报错，告诉我们什么地方出错了。由于我们把这条语句放在 app.js 头部，浏览器仅仅会为我们的代码和这条语句之后加载的代码开启严格检查模式。

添加路由函数

现在，我们的测试用例还是失败了，但是由于我们已经为应用创建了命名空间，所以显示的错误信息有所不同。

```
1 spec, 1 failure
LearnJS can show a problem view
  TypeError: learnjs.showView is not a function
```

在这里，测试结果显示 learnjs.showView 不是一个函数。没错，它是还没有定义[6]，因为我们还没有创建它。为了使测试通过，我们还有两个步骤要执行，这就是其中的一个。既然测试结果说无法找到 showView 函数，那么我们就来解决这个问题。

showView 函数将是我们要添加到应用命名空间的第一个函数。在这个函数里，我们将用 jQuery 选择 view-container 元素，并且在它后面添加视图的标记。由于我们还没为这个题目视图创建过独立的函数，所以先硬编码这个函数来插入一个 problem-view 类的 标签，因为这正是测试用例所要求的。将 showView 的实现添加到 app.js 的命名空间中：

learnjs/2100/public/app.js

```
'use strict';
var learnjs = {};
learnjs.showView = function(hash) {
  var problemView = $('<div class="problem-view">').text('Coming soon!');
  $('.view-container').empty().append(problemView);
}
```

Joe 问：

我真的要这样创建视图标记吗？

虽然通过 jQuery 动态创建标记是可能的，有的时候还很有效，但是只用 jQuery 创建大量的标记就会有些枯燥。showView()的第一种方式就是这样的，但这种实现并不会保留很久。

当这些都添加了之后，测试显示的错误信息又变了：

```
1 spec, 1 failure
LearnJS can show a problem view
```

6 http://eloquentjavascript.net/01_values.html#h_WAVjYN+DYj

```
Expected 0 to equal 1.
```

现在要避免往题目视图中添加任何其他东西，稍后（和测试同时）再给它添加功能。现在，你需要专注于开发路由器的功能。好消息是，你又近了一步，因为测试用例的失败结果正如你所预期的那样。

创建视图容器

在此之前，测试用例一直报 JavaScript 错误。现在它确实是断言失败。所有代码执行都没有错，仅仅是测试用例想要一个元素，但是没找到。测试用例报的这种错误很明确，能指导我们下一步该怎么做。

这种情况下，即使 `showView` 函数打算创建并添加标记，但是在页面中没有视图容器来插入标记。所以，当我们调用 jQuery 的 `append` 方法时，标记无处可以添加。要修复这个问题，我们需要对标记做两个改动。首先必须创建视图容器，其次我们需要在应用中引入一个新的元素，使我们的测试用例能在应用中访问标记。

目前，测试用例还没有办法在应用中访问标记。`test/index.html` 中的测试执行器与我们在 `index.html` 中的应用不是同一个文档。不过，有办法将它们俩放一起，但是需要先在应用中添加一些东西。

在预备的开发环境中，测试执行器有一个名为 `public/tests/SpecHelper.js` 的文件。这个文件有一段代码，能复制我们应用中带有 `markup` 类的元素中的任意标记。它把标记插入到测试执行器的页面中，让所有测试用例都能获取。为了使我们的视图容器和应用的所有标记与测试用例区分开，你需要创建这个元素。

同时，你还需要把 `view-container` 类添加到包含 Skeleton `container` 类的`<div>`中去。我们把它作为视图容器。既然内容将被放到容器中，一旦我们把视图容器作为 Skeleton 容器，我们的视图就可以基于 Skeleton 网格的行和列创建它们的标记。

完成这两项改造后，应用的 `index.html` 文件中的`<body>`将会如下所示：

learnjs/2100/public/index.html

```
<body>
  <div class='markup'>
    <div class='view-container container'>
      <div class='row'>
        <div class='one-half column'>
          <h3>Learn JavaScript, one puzzle at a time.</h3>
          <a href='' class='button button-primary'>Start Now!</a>
        </div>
        <div class='one-half column'>
          <img src='/images/HeroImage.jpg'/>
        </div>
      </div>
    </div>
  </div>
</body>
```

保存完更改之后，执行我们的测试脚本：

```
1 spec, 0 failure
LearnJS can show a problem view
```

现在测试通过了！到这里，我们完成了一些重要的步骤。我们编写了第一个自动化测试脚本，并通过了测试。我们创建了一个路由函数，使应用能在两个视图间跳转而不需要重新加载页面。我们创建了一个视图容器，所有视图能用它来装载它们的标记。最后，我们还使所有的应用标记对测试可见，方便我们测试视图和它们的 UI 元素。

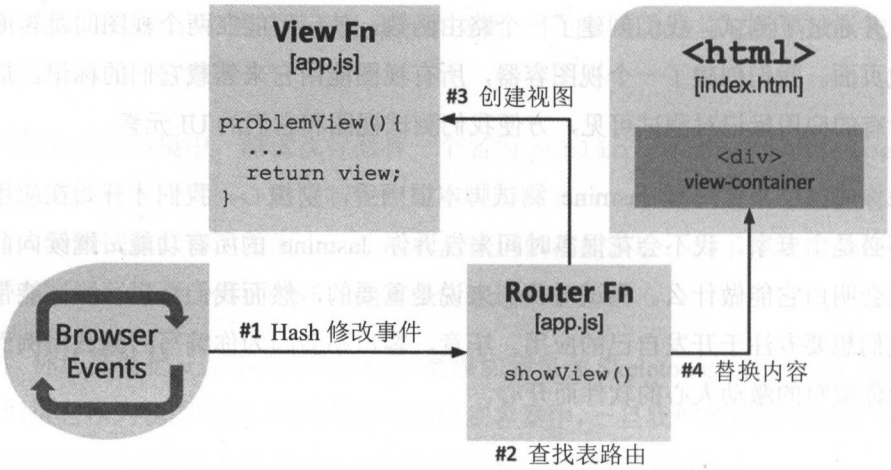
如果你现在还不觉得写 Jasmine 测试脚本很顺手，别担心。我们才开始在应用中使用它，你不必是个专家。我不会花很多时间来告诉你 Jasmine 的所有功能，继续向前吧，慢慢地你就会明白它能做什么。测试对我们来说是重要的，然而我们希望写测试能带来更多价值，我们想要专注于开发自己的应用。毕竟，客户不会因为你编写的测试用例而开心，而是因为你编写的激动人心的软件而开心。

添加路由

我们的路由器现在能显示第二个视图了，但也仅此而已。showView()函数只是通过硬编码来显示题目视图。为了解决这个问题，我们要想办法让路由器把 hash 值和视图函数关联起来。这种关联我们称之为路由（route），现在我们就为应用添加一个路由。

我们的路由将用一个 JavaScript 对象来表示，它是一个 hash 值查询表。这个对象维系 URL hash 值和视图函数之间的关系。视图函数返回一个包含视图标记的 jQuery 对象。尽管我们能添加更多记录到这个 routes 对象中，用这种方式来为应用添加更多视图，但是我们现在只需要一个路由。

添加这个 routes 对象之后，我们能将视图的创建行为如数剥离出来，添加到独立的函数中，如下图所示。现在路由器的工作需要 4 步。Hash 修改事件还是会触发路由函数 showView，但是路由器现在调用解耦的视图函数来创建插入到页面中的视图。能将视图创建逻辑抽取出来并封装成函数，意味着我们不通过路由器就能测试它，如果使用路由器的话会使测试变得很复杂。



为了引入这些路由，需要创建另一个测试来驱动开发。如果能直接测试 routes 对象，

将使这段代码日后难以重构。例如，如果只是修改了 routes 对象的结构而没有修改路由器的行为，我们就不希望测试会失败。

可以这么做，写一个使用现成 showView 路由函数的测试用例，来断言根本没有 hash 值的时候会发生什么。这种测试有时候被称作 null case 或者 default case。这不是说我们将使用 null 这个 JavaScript 关键字。它指的是一种常见的不使用数据的交互。空字符串、空对象、空数组和数值 0 经常出现在 null case 测试中。

在这个测试用例中，我们要测试当 hash 值是一个空字符串的时候，应用是否会显示着陆页视图。这个测试中的方法和第一个测试很相似：尝试通过 jQuery 选择一个元素，断言的确有一个元素被正确地选中。

learnjs/2200/public/tests/app_spec.js

```
describe('LearnJS', function() {
  it('can show a problem view', function(){
    learnjs.showView('#problem-1');
    expect($('.view-container .problem-view').length).toEqual(1);
  });
  it('shows the landing page view when there is no hash', function() {
    learnjs.showView('');
    expect($('.view-container .landing-view').length).toEqual(1);
  });
 });
```

为了通过这个测试，我们可以引入一个 routes 对象和名为 problemView 的视图函数。路由器将根据这个 routes 对象找到一个视图函数，如果没有 routes 对象，应用将返回到着陆页。因为着陆页被默认添加到视图容器中，意味着如果 hash 值没有匹配到任何路由，也就等于我们什么都不能做。

learnjs/2200/public/app.js

```
'use strict';
var learnjs = {};
learnjs.problemView = function() {
  return $('<div class="problem-view">').text('Coming soon!');
}
```

```
learnjs.showView = function(hash) {
  var routes = {
    '#problem-1': learnjs.problemView
  };
  var viewFn = routes[hash];
  if (viewFn) {
    $('.view-container').empty().append(viewFn());
  }
}
```

最后，你需要将着陆页视图用 landing-view 类包裹在一个 中，这样我们就能在应用和测试用例中选择它。

learnjs/2200/public/index.html

```
<body>
  <div class='markup'>
    <div class='view-container container'>
      <div class='landing-view'>
        <div class='row'>
          <div class='one-half column'>
            <h3>Learn JavaScript, one puzzle at a time.</h3>
            <a href='' class='button button-primary'>Start Now!</a>
          </div>
          <div class='one-half column'>
            <img src='/images/HeroImage.jpg'/>
          </div>
        </div>
      </div>
    </div>
  </div>
</body>
```

这样一来，我们就通过了两个测试。现在单击着陆页中的按钮还是不能改变视图，但是稍后，我们就能添加一个带有 hash 值的 href 属性来控制跳转。一旦应用开始监听 hash 事件，如果路由器能根据 hash 值匹配到路由，就将调用合适的视图函数来构建视图，用这个视图的标记替代 view-container 元素。如果它无法匹配到一个路由，就什么也不做，就地留在着陆页。

添加视图参数

虽然我们已经为应用添加了第一个路由，但是还有一些地方不对劲。这个路由关联到一个特定的题目视图——problem #1。我们不希望为应用的每一个题目视图添加一个新路由，而希望通过一个路由（和一个视图）来显示应用中的所有题目。

为了达到这个目的，我们把视图参数化。将 hash 值分成两部分：一个名字和一个参数，中间用一个短横线（-）作为分隔符。调用该视图函数时将传递参数，视图函数可以用这些数据来构建视图。我们把这些值称为视图参数（view parameter）。它们是在应用中给视图传递数据的第一种方式。

在这个特例中，视图参数是题目的 ID。但是在开发应用时，可以把任何值作为视图参数。我们可以传递另一些标识符，例如数据库主键、UUID 或者加密的 hash 值。我们还可以传递原始数据、明文或者加密的 JSON 或 Rison 格式[7]的数据。路由器不会规定视图参数是什么，以及如何拦截或者如何解析它们。识别接收了哪种参数和怎么使用这些参数，是视图的职责。

为了测试这个功能，我们将引入一种新的测试结构，称作 spy（间谍）。spy 被用于测试两段代码之间的相互作用。在这里，我们需要测试路由器和我们为题目视图新建的视图函数之间的调用情况。

用 spy 测试调用

spy 是一种特定的测试替身（test double）。测试替身代替真正的对象或者函数帮助我们进行测试。一些测试替身能对它们是怎样被使用的（或不被使用）做出断言，另一些测试替身则能简单地模仿其他一些东西。Jasmine 内建了对 spy 的支持，它能模仿函数，让我们验证想要测试的代码是如何与测试作用域之外的代码相互作用的。

7 https://github.com/Nanonid/rison

听测试的话

你可能经常听经验丰富的程序员讲，代码的组织方式应该是能把“一件事做得很漂亮”，但到底什么是“一件事”？这很难准确地计量，但是测试用例为我们提供了一种方法。和路由器一样，编写测试往往能指导设计。如果一个测试的结构太长而难以阅读，也许你所测试的这个功能到底如何就需要靠它证明自己了。

我们将用 spy 来验证路由器和题目视图函数之间的相互作用。我们要断言当调用路由函数的时候，路由器正确地提取了视图参数，并将其传递给视图函数。我们用 Jasmine 的 spyOn 函数来实现，spyOn 函数能用一个 spy 临时替代指定的函数。这个 spy 将记录下对它的所有调用，我们能用特定的 Jasmine 匹配器，例如 toHaveBeenCalledWith，来确定 spy 是通过特定的参数调用的。一旦测试结束，Jasmine 用原来的函数替代 spy，因而其他测试用例不受影响。用 Jasmine 的 spy，我们能写出像下面这样的测试脚本：

learnjs/2300/public/tests/app_spec.js

```
it('passes the hash view parameter to the view function', function() {
  spyOn(learnjs, 'problemView');
  learnjs.showView('#problem-42');
  expect(learnjs.problemView).toHaveBeenCalledWith('42');
});
```

在这段测试脚本中，我们通过传递到 spyOn 的两个参数来配置 spy。第一个参数是函数所在的对象。在这个例子中，它就是我们的命名空间。当然，你也可以传递 window 对象来监视全局函数。第二个参数是该函数的名字，它被作为一个字符串传递。注意，当你做这个断言的时候，Jasmine 要求你直接引用这个函数，因为它已经被一个 spy 替换。

确认这个测试用例报错失败，而且错误信息为“预计 problemView spy 会通过['42']被调用，但是并没有”以后，我们开始准备改进 showView 来支持视图参数。将 hash 值分割为一个数组，然后用这个数组中的一部分值，来找到正确的路由并构建视图。

learnjs/2300/public/app.js

```
learnjs.showView = function(hash) {
  var routes = {
```

```
➤    '#problem': learnjs.problemView
  };
➤  var hashParts = hash.split('-');
➤  var viewFn = routes[hashParts[0]];
  if (viewFn) {
➤    $('.view-container').empty().append(viewFn(hashParts[1]));
  }
}
```

这足以使我们的测试通过了。现在，题目视图不会根据传递过来的参数做任何事情。它还没有任何内容，我们也还没准备好添加任何内容。接下来，我们将使用 problemView 函数中这个视图参数。

通过使用 spy，在不改变视图的前提下，我们已经成功测试了路由器和视图之间的调用。着重测试调用，而不是流程，我们才能逐个测试类似的功能。即使题目视图对视图参数不做任何处理，我们也能确保它被正确地传递了。

处理视图函数中的参数

为了完成测试的另一半，下面开始测试 problemView 函数。到目前为止，这个函数中的代码仅仅通过路由函数测试过。对于一个测试而言，这样做是没有问题的，但是继续像这样测试题目视图，将使测试包无法提供全面的信息。

构建一个 FIRE 测试套件

为应用编写测试套件时，我会让某些属性值保持不变。我希望这个测试套件是快速（Fast）、信息丰富（Informative）、可靠（Reliable）以及完备（Exhaustive）的，简称为 FIRE。例如，测试脚本是快速的，意味着每次对代码做修改就能运行一次测试；保证它们是可靠的，意味着它们不会随随便便中断。

在本章，我把一个看似独立的测试分为了两部分。这么做是为了让测试能提供更多信息。如果哪里出现问题，希望它们能清楚地解释。一个测试脚本只验证应用的某一个功能，有助于确保被引入到应用的 bug 只会导致一个测试失败。如果这些测试命名贴切

而且组织结构合理，那么很快就能找出失败的原因。

要不然，如果为应用中每一个端到端的工作流都写测试，那么对于某一个修改可能有几十个测试要做。然后，如果我引入一个 bug，或者改变了什么，这时候就不是一个测试来通知我某处出现问题了，而是一堆失败的测试告诉我到处都有问题。解决问题的过程就变成一件十分无聊的工作，因为必须蹚过全部的失败测试，搞清楚发生了什么事，需要怎么做。

为了验证一切都是测试过的，我需要确保测试套件是完备的。既然没有测试端到端的每个工作流，那就要对工作流的各个部分进行独立测试，然后对它们之间的交互进行测试。这样我才有信心构建能快速运行的有效的测试套件。

为了使我们的题目视图的测试覆盖到应用的更多功能点，应该为视图新建一个 describe 块，把 it 函数放在外层 describe 里。Jasmine 允许用这种方式嵌套 describe 函数，来给一部分测试提供上下文和作用域。当我们添加测试时，它应该是下面这样：

learnjs/2400/public/tests/app_spec.js

```
describe('LearnJS', function() {
  it('can show a problem view', function() {
    learnjs.showView('#problem-1');
    expect($('.view-container .problem-view').length).toEqual(1);
  });
  it('shows the landing page view when there is no hash', function() {
    learnjs.showView('');
    expect($('.view-container .landing-view').length).toEqual(1);
  });
  it('passes the hash view parameter to the view function', function() {
    spyOn(learnjs, 'problemView');
    learnjs.showView('#problem-42');
    expect(learnjs.problemView).toHaveBeenCalledWith('42');
  });
  describe('problem view', function() {
    it('has a title that includes the problem number', function() {
      var view = learnjs.problemView('1');
      expect(view.text()).toEqual('Problem #1 Coming soon!');
    });
```

```
  });
});
```

如果看到一个失败的测试，可以添加这样的实现来使其通过：

learnjs/2400/public/app.js

```
learnjs.problemView = function(problemNumber) {
  var title = 'Problem #' + problemNumber + ' Coming soon!';
  return $('<div class="problem-view">').text(title);
}
```

在下一章，我们将利用传递过来的视图参数优化视图。现在，我们有一件更重要的事要做：加载应用，这样就能打包并部署一个新的（略微）优化版本。

加载应用

和所有的 Web 页面一样，我们的单页应用也需要加载。现代浏览器天生擅长从互联网下载内容，简单来说就是：当你输入一个 URL，浏览器获取并下载 URL 指定的文件。一旦文件被下载下来，如果它是 HTML 文件，浏览器将开始下载标记指定的其他资源文件，例如 CSS、JavaScript、字体、图片或者视频。

Joe 问：
怎样做测试?

你可能注意到了，我没有讲解如何为 index.html 中<script>标签中的代码写测试。我还会讲吗？当然会，但不是使用到目前为止讨论的这些技术。不过，我要是跳到这部分，解释一下要怎样做，你肯定会发现自己不知道怎样才能有效地测试应用的功能。不是这些测试太复杂没什么用，就是你根本不知道如何做测试。特别是当你还在学习如何用测试驱动设计时，这一点会十分明显。理解如何处理这种情况对于有效地使用测试是十分重要的。

要让自动化测试覆盖到每一行代码，实在没有什么必要。像这样教条式的声明是站不住脚的。准确的判断以及一些合理的指南通常对你更有用。下面列出几条：

- 不要认为有特殊的实现。这样会使测试更难做。做事灵活一点。
- 可测试性比封装性更重要。有后者固然不错，然而前者是基本要求。
- 假设还有别人知道如何测试，你会怎样找出他们的测试方法？
- 在修改未测试的代码之前，先为它们写好测试。或许你会发现某个新方法。
- 如果所有测试都失败，把未测试的代码与可测试代码分开。

我们的应用像网页一样加载，但是它像原生应用一样只需要加载一次。然而，当浏览器渲染完这个页面时，单页应用的加载过程还几乎没完成，还需要添加事件监听器，初始化数据结构和加载其他资源文件。有一个办法可以监视浏览器的加载进度，并做一些应用要求的额外操作。

通过在页面顶部的<script>标签里添加一些 JavaScript 代码，就可以实现这一点。但是这么做会有一个问题：浏览器会尽可能快地执行这段 JavaScript 代码，甚至早于浏览器渲染 HTML 元素。这可能会使我们无法通过修改或添加元素来创建视图，因为这些元素彼时在页面里还不存在。

要避免这个问题，有一种方法就是添加一个监听器，当页面准备就绪时，监听器就能知道。我们也可以在<body>元素的底部插入<script>标签，绝大部分浏览器会在加载完页面之后执行里面的代码。在这个例子里，我们要开发双保险路由，完成这两件事。在页面底部添加一个<script>标签，并设置一个事件监听器。我们用 jQuery 的$.ready()方法来实现。

响应事件

$.ready()方法和其他 jQuery 事件方法一样，允许为事件添加一个监听器。只需要传递一个回调函数作为参数，然后当该事件被触发时，你的回调函数就会被调用。在这个例子中，这个事件是 DOMContentLoaded，一旦所有的 HTML 被加载完成，浏览器将马

上触发 DOMContentLoaded 事件。

回调函数的名字为 appOnReady，它将监听应用的加载情况。为了使这个函数监听加载事件，我们将调用$.ready()，并传递 learnjs.appOnReady 作为参数。将这段代码添加到位于页面<body>元素底部的<script>标签中，像下面这样：

learnjs/2500/public/index.html

```
<body>
  <div class='markup'>
    <div class='view-container container'>
      <div class='landing-view'>
        <div class='row'>
          <div class='one-half column'>
            <h3>Learn JavaScript, one puzzle at a time.</h3>
            <a href='' class='button button-primary'>Start Now!</a>
          </div>
          <div class='one-half column'>
            <img src='/images/HeroImage.jpg'/>
          </div>
        </div>
      </div>
    </div>
  </div>
➤ <script type='text/javascript'>
➤   $(window).ready(learnjs.appOnReady);
➤ </script>
</body>
```

需要注意的是，我们的 标记没有把<script>标签包含在内。另外，当测试执行的时候，<script>标签将会被执行，然后我们的应用将会被加载。我们希望，当应用在测试中被加载时是由我们控制的。另外，加载应用的过程中，任何状态的变化，例如当前页面的变化，都将影响测试结果。

加载页面时，一旦该页面准备就绪，jQuery 就会调用 learnjs.appOnReady()，这样我们的应用能做一些必要的准备工作。注意，当你把 appOnReady 传递给$.ready()

时是不带括号的。此时并没有调用它，只不过是引用了它[8]。如果不小心加上了这些括号，应用将出现意想不到的行为，因为它无法在对的时间加载。

接下来，要把 learnjs.appOnReady() 函数加到应用里。我们希望这个函数能在页面加载的时候调用路由函数。这样，就能用 spy 来断言 showView 是否是用当前页面的 hash 调用的。

learnjs/2500/public/tests/app_spec.js

```
it('invokes the router when loaded', function() {
  </div>
  spyOn(learnjs, 'showView');
  learnjs.appOnReady();
  expect(learnjs.showView).toHaveBeenCalledWith(window.location.hash);
});
```

现在我们可以把函数添加到应用的命名空间里。把这段代码放在 app.js 文件底部。

learnjs/2500/public/app.js

```
learnjs.appOnReady = function() {
  learnjs.showView(window.location.hash);
}
```

这样我们就能通过测试了。现在我们需要订阅 hashchange 事件，而且路由器已经准备好做测试了。

响应 hash 事件

我们已经定义了路由，并且可以创建视图了。现在可以给 hashchange 事件添加监听器。当然，这得从新的测试用例开始。

我们将用 jQuery 的 trigger 函数来触发 hashchange 事件，而不是在测试中改变 hash 值。这么做的原因是，测试脚本和应用一样在浏览器中运行，如果改变测试程序的 hash

8 http://eloquentjavascript.net/03_functions.html#h_y6WGSsYfER

值可能会出问题。至少它会使我们的测试脚本的状态异常，因为这个 hash 值可能和与执行的测试用例所预期的不同。这可能会使我们的测试脚本不可靠，应尽量避免。

为了触发事件，我们将加载应用并且监视 showView 函数。加载应用之后监视这个函数，可以确保我们的 spy 记录下因 hashchange 事件而导致的 showView 调用，而不是仅仅在应用被加载时的 showView 调用。在此之后，我们将调用 trigger 函数，传入想要触发的事件的名字，在这里就是 hashchange。然后，我们就能断言 spy 是由正确的参数调用的。

learnjs/2600/public/tests/app_spec.js

```
it('subscribes to the hash change event', function() {
  learnjs.appOnReady();
  spyOn(learnjs, 'showView');
  $(window).trigger('hashchange');
  expect(learnjs.showView).toHaveBeenCalledWith(window.location.hash);
});
```

既然我们的测试失败了，那么可以为 appOnReady 添加一些功能来让该测试通过。为了注册事件监听，我们将把一个函数赋值给 window 对象的 onhashchange 属性。浏览器 API 提供了这个属性，我们可以用它为 hashchange 事件注册事件处理程序。这个函数将在 hash 值改变的时候被调用。我们就在 appOnReady() 函数中注册这个事件监听器。

learnjs/2600/public/app.js

```
learnjs.appOnReady = function() {
  window.onhashchange = function() {
    learnjs.showView(window.location.hash);
  };
  learnjs.showView(window.location.hash);
}
```

在这个函数中，可以看到我们添加了一个事件监听器。在这个监听器里，我们调用 showView() 函数，传递 window.location.hash，和加载应用时做的一样。一旦我们完成改造，新的路由器应该能正常工作了。为了确保它在应用中能正常工作，我们将为<a>元素添加一个属性，用它在着陆页调用“启动”按钮。我们可以为题目视图以 hash 值的形

式添加一个链接。这样，当用户单击这个按钮时，它们将被带到第一道题的视图中去。

learnjs/2600/public/index.html

```
<a href='#problem-1' class='button button-primary'>Start Now!</a>
```

既然我们已经添加了 hash 变化的监听器和一个链接，我们可以试一下应用，看看它的实现是否正确。从着陆页进入，然后单击“启动”按钮。之后应该显示题目视图。如果是这样，我们就知道路由器起作用了，而且还知道不给应用服务器发请求应用也能改变视图。

接下来，我们开始部署现有的代码。在上一个版本中，只有一个着陆页，它只有一个什么也不做的启动按钮。现在这个版本稍微好一些。不过有些功能还是没有。比如，不能单击“回退”按钮使其返回着陆页。但是我们马上就能解决这些问题了。虽然还不能将应用分享给潜在用户，但每次优化应用以后，我们都希望可以尽快部署。如果相信我们的应用能如期望的那样工作，观察它在生产环境中是否真的可以运行，是让自己确信的唯一方法。

再次部署

我们已经添加了路由器，并且应用能响应对“启动”按钮的点击。我们还有办法添加新的路由，并且有一套完美的测试脚本来证明它们全部都正确运行。接下来，再次使用我们在第 1 章中用过的 `./sspa deploy_bucket` 命令来部署应用。完成这些工作之后，在考虑下一步做什么的时候，在浏览器中打开应用体验一下。

下一步

现在你已经理解了路由器的工作机制，你可能想进一步了解其他主题。由于篇幅所限，本书不会详细介绍它们，此处给出一些简短的说明，你可以参考相关资料自行学习。

Jasmine 匹配器

Jasmine 支持所有类型的匹配器，它们是你用 `expect` 做断言后所调用的函数。和

结构化的 `describe` 和 `it` 函数不同，匹配器有很多种，你可能得花点时间来学习怎么使用。如果要了解更多关于 Jasmine 匹配器的知识，可以访问 http://jasmine.github.io。本书使用的是 2.0.2 版本。

Jasmine-jQuery

Jasmine 是一个很好的测试框架，并不局限于 Web 开发。作为 Jasmine 提供的内置匹配器的补充，Jasmine-jQuery 扩展添加了许多 HTML 指定的匹配器，比如 `toExist` 和 `toHaveClass`，它们使 Web 应用的测试变得更简单。你可以在 https://github.com/velesin/ jasmine-jquery 上了解更多关于 Jasmine-jQuery 的内容。

测试替身

我们在这一章中提到了怎么使用 Jasmine spy，对于不同的用途其实有很多种测试替身。你可以在这里看一些例子：http://xunitpatterns.com/Test%20Double.html。

路由库

我们的路由需求有点简单，但是如果你的需求比较复杂，也许会考虑使用路由库。Director（https://github.com/flatiron/director）和 Page.js（http://visionmedia.github.io/page.js/）是路由库的例子。如果你发现需要对我们这个简单的路由要做很多修改，或许使用路由库是一个不错的选择。

JavaScript 可选的测试框架

在本书中我们选择 Jasmine 作为测试框架。市面上至少有一打其他选择，各有优劣。QUnit（http://qunitjs.com/）是一种受欢迎的框架，另一个是 Sinon.js（http://sinonjs.org/）。Vows.js（http://vowsjs.org/）是一个异步的测试框架，在这个问题上有独到的见解。对于本书中讨论的技术，可以用这些框架中的任一款。对它们做一些研究，再决定哪一个更适合你吧。

Hash改变事件

本书中我们只讲了hashchange事件的一点皮毛。你可以在这个事件回调函数[9]中获取到旧的hash值。你也能通过调用window.history下的pushState和popState来操作浏览器历史记录。这些将有益于模拟更多传统的路由方法。

下一章，我们将实现题目视图。我们要来看看如何用HTML模板来替换现有的jQuery生成的标记。我们的应用将拥有更多功能，最终部署一个值得分享给用户的版本。

9 https://developer.mozilla.org/en-US/docs/Web/API/WindowEventHandlers/onhashchange

第 3 章
单页应用的必要组件

无论评论家们说什么，现代浏览器毫无疑问是一种应用容器（application container），至于它是否够好，那就是另一个话题了。Web 标准，比如 HTTP、HTML 和 JavaScript，是可以运行在全世界成千上万台计算机、手机和平板电脑上的应用分发平台的基石。现在你手头可能就有其中至少 2 种设备。开发 Web 应用是让你的应用能被最多用户使用的最快方式。为了跟上这股席卷全球的趋势，我们需要弄明白怎样才能让浏览器发挥潜能成为最好的应用容器。

既然我们已经有一个可测试的路由器，就可以在此基础上继续开发我们的应用了。在本章中，我们将为应用加入其他必要组件，在你开发的所有单页应用中（无服或者其他类型），都能发现它们的身影。同时我们将开发应用的视图、创建简单的数据模型，然后把模型中的数据绑定到标记中。接下来，我们要看一看如何利用视觉效果来增强用户输入的反馈，而且还会构建一个带导航条的应用外壳。到本章结束时，我们就会得到一个功能完备的 Web 应用。

首先，应该对这个应用的外观有一个初步的想法。前面已经做了不少准备，只是还没理清头绪。我们已经知道要开发的是一个 JavaScript 编程解题应用，但是没有确定这个应用要怎么运作，与用户怎么交互。是时候考虑一下这些问题，给出一个设计方案了。

创建视图

在开发应用时，用户界面设计可能是比较有挑战性的任务。为了设计出用户能轻松使用的界面，你需要理解他们的期望和目标。从用户的角度出发思考他们到底想要什么，不是一件容易的事，但如果面前有一个能运转起来的半成品应用，这件事就会变得简单得多。因为已经有的那些功能会给你灵感，而缺失的功能则会引起你的注意。

就像一图胜千言一样，一个能运转的原型胜过千次小组讨论。

我们的应用中，题目视图要求用户“填空”使 JavaScript 函数返回一个真值。呈现给用户的题目应该像下面这样：

```
function problem() {
  return 2 + __ === 4;
}
```

我们的用户会尝试搞清楚 JavaScript 应该在_的地方输入什么才能使 `problem()` 函数返回 `true`。如果他们填了空，程序就会判断其填入的内容是否正确。这意味着，每个问题可能会有很多不同的正确答案。设计这个视图的时候，我们需要考虑这个工作流有哪些必要的元素。

这个视图是关于编程题目的，所以我们需要有一种方式来展示这道题。为了给用户提供一些上下文，就要展示标题，其中包括题目的序号。一条关于题目的描述会帮助用户正确理解他们需要做什么。为了让用户验证自己的答案，还需要提供一种方式使他们能输入解题的代码。本章我们把所有这些元素放到一起来构建应用的主视图——在视图中逐步添加这些元素，同时验证我们关于工作流的假设是否正确。

为了优化我们的题目视图，首先抽取出视图的标记作为 HTML 模板。HTML 模板是一个管理标记的很棒（且简单）的方法。它们灵活、易于测试，且不需要特定的库或者工具。如果要用标准的 HTML 编辑工具和格式化函数来创建标记，使用模板比用 jQuery 编程实现要简单得多。一旦创建了模板，就可以添加一个`<title>`元素，用它来显示标题（而不是

向视图插入文本）。创建视图模板之前，需要在应用中提供一个地方让模板运行。

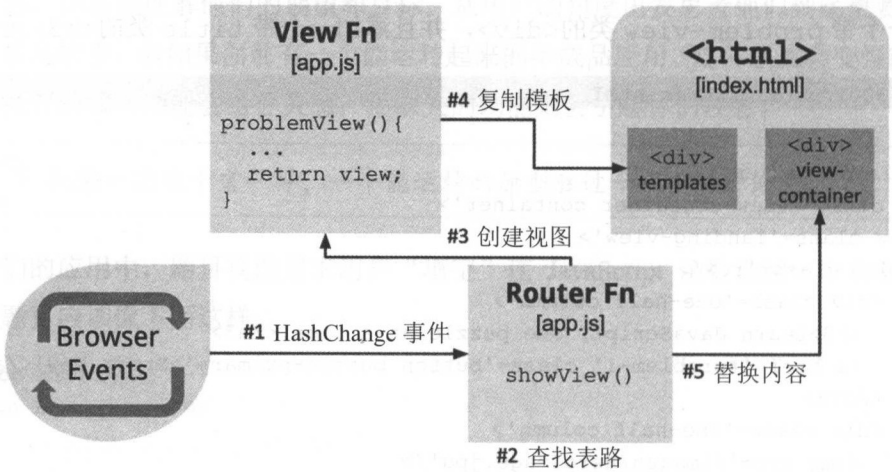
为了给应用的模板提供空间，先要在应用标记中创建 一个新的 并加上 templates 类。我们将用这个元素来承载全部的模板。在这个 templates 中，再创建一个带 problem-view 类的 ，并且添加一个带 title 类的<h3>元素。

learnjs/3001/public/index.html

```
<body>
  <div class='markup'>
    <div class='view-container container'>
      <div class='landing-view'>
        <div class='row'>
          <div class='one-half column'>
            <h3>Learn JavaScript, one puzzle at a time.</h3>
            <a href='#problem-1' class='button button-primary'>Start Now!</a>
          </div>
          <div class='one-half column'>
            <img src='/images/HeroImage.jpg'/>
          </div>
        </div>
      </div>
    </div>
➤   <div class='templates'>
➤     <div class='problem-view'>
➤       <h3 class='title'></h3>
➤     </div>
➤   </div>
  </div>
  <script type='text/javascript'>
    $(window).ready(learnjs.appOnReady);
  </script>
</body>
```

接下来，需要在 problemView()函数中创建此模板的副本。复制了它，就可以一遍又一遍地重复构建这个视图，而无须担心一个实例的状态影响另一个。从 temlates 中挑出 problem-view ，然后用 jQuery 的 clone()方法来复制。在本章后面的升级版路由器中可以看到这个复制操作。

一旦完成复制，就要更新<title>元素的文本。可以通过 jQuery 选择<title>元素，然后用题目的序号来构建标题的文本。完成之后，通过视图函数返回视图的标记。经过所有这些改造之后，题目视图函数应该看起来像下图这样。



learnjs/3001/public/app.js

```
learnjs.problemView = function(problemNumber) {
  var view = $('.templates .problem-view').clone();
  view.find('.title').text('Problem #' + problemNumber);
  return view;
}
```

你需要做的最后一件事是，隐藏 templates 。我们不希望模板出现在应用里，写几句 CSS 就可以隐藏它们。为 index.html 的<head>中内联<style>元素添加一条规则，如下：

learnjs/3001/public/index.html

```
<style type="text/css" media="all">
   body { margin-top: 30px; }
  .templates { display: none; }
</style>
```

现在，如果你打开应用并单击“启动”按钮，将跳转到题目视图并显示一个简单的标

题，如下图所示。

Problem #1

像这样使用模板来创建视图，标记的管理更加简单，同时应用也十分灵活，能动态显示我们想要的内容。jQuery 使我们不需要额外的模板库就能轻易地复制、改变和插入这些模板，稍后你会发现在应用的其他部分也可以复用这项技术。

继续完成题目视图。我们可以添加题目描述、题目代码和用户提交答案的地方。在添加这些之前，先要搞清楚所有这些数据从哪里来，怎样才能使我们的应用获取它们。总之一句话，我们需要一个数据模型（data model）。

Joe 问：
要怎么做测试？

我认为在设计时考虑代码的可测试性是非常有必要的，而且最好的方式就是先写测试用例。但本书的重点不是测试，因此，我还是会演示使用了新的或高级的技术的测试脚本，但不会要求你也写类似的测试脚本。

不过本书接下来的代码就是测试脚本。否则，你对开发这类 Web 应用没有一个完整的概念。设计一个当下能运行的软件是一回事，而设计一个做了修改之后仍然能运行的软件则是另一回事，显然后者更具有挑战性。你可以在 pragprog.com 上查看所有的测试脚本（包含代码）。如果你对某一段代码是如何测试的有任何疑问，在那里能找到答案。

定义数据模型

在开发 Web 应用的时候，用面向对象的 JavaScript[1]来定义数据模型是通用的做法。首

1 http://eloquentjavascript.net/06_object.html

先，要创建 JavaScript 类（实际上就是一些函数），它们使用原型链继承（prototypical inheritance）[2]来为你的问题建模。然后创建这些对象的图，来存储数据以及执行对数据的操作。如果在应用里使用这种方法，我们很可能用几个 `Problem` 对象作为结束，它们包含一个 `checkSolution` 函数，此函数接受其他对象类型（可能是 `Solution`）作为参数。这两种对象类型会继承另一个对象类型（可能是 `Entity` 或者 `Model`）的持久化和序列化的行为。

现在我们使用另一种方法，仅用数组和对象这两种 vanilla 数据结构存放数据，不使用用原型行为扩展的 JavaScript 对象图来定义数据模型。我们在应用里创建一个函数目录来操作数据。

我们希望通过这种方式实现较低的对象映射阻抗（object mapping impedance）。对象图数据模型有一个限制，即必须不断地把它们映射到底层的数据存储中。例如，如果应用中使用了一个关系型数据库，就可能需要用一个对象关系映射（ORM）工具来将数据序列化到数据库，以及反序列化从数据库读取的数据。

如果使用原型继承的数据模型，在尝试序列化这些对象时可能会遇到问题。JavaScript 对象是不区分数值和行为的。函数不过是对象的属性而已，和其他任何属性一样，例如字符串和数字。如果在模型中把数据和行为混在一起，当我们将它们序列化到数据库或者从数据库读取出来时还是要把它们分离。最好在数据模型对象和数据库之间有一个更直接的映射，那么我们就不需要做这个分离工作了。

在第 5 章，你会看到我们用来承载题目数据和用户数据的 vanilla JavaScript 数据结构直接映射为数据库存储的记录。DynamoDB 支持将 JSON 文档存储为一条条的记录，因此选择一个可以被简单序列化为 JSON 的数据模型，意味着我们不需要为了把数据转换成数据库可以理解的格式而做额外的工作。这是一个把 Web 标准和 Web 服务结合起来简化开发的典型例子。

---

2 http://eloquentjavascript.net/06_object.html#h_SumMlRB7yn

创建一个对象数组来盛放题目相关的数据。每个对象将包含应用中每道题需要的属性。在 app.js 文件的开头（在函数之前，命名空间之后）定义这个数组。你可以在这两个例子中看到对象的格式，而且可以随意添加更多属性。

learnjs/3100/public/app.js

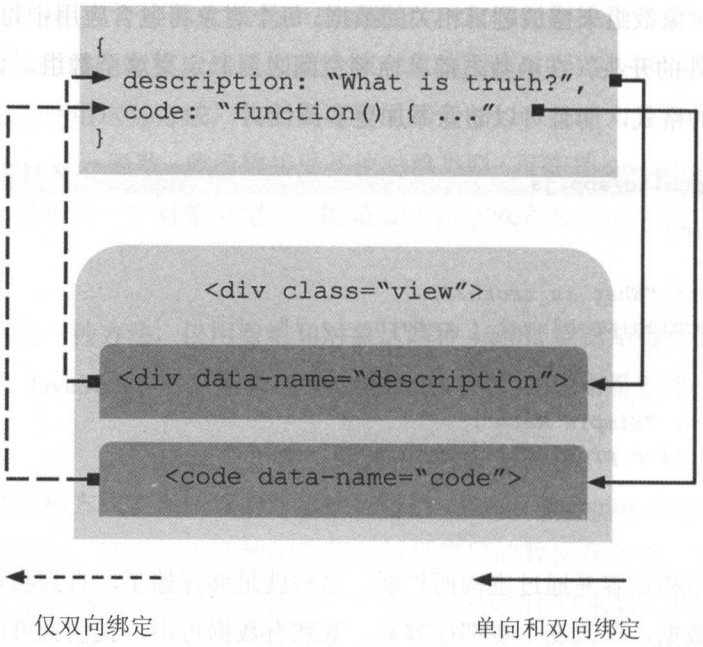
```
learnjs.problems = [
  {
    description: "What is truth?",
    code: "function problem() { return __; }"
  },
  {
    description: "Simple Math",
    code: "function problem() { return 42 === 6 * __; }"
  }
];
```

这个数据结构很容易通过重构而扩展。这些就足够开始了，直到我们开始持久化（或者引用）这些数据，要改变主意都很容易。既然有数据可用，我们就可以研究如何把它们绑定到视图的标记中，然后为题目视图添加更多元素——题目本身的描述和代码。

数据绑定

有了数据模型，我们需要找到把数据引入标记的方式。最简单的就是像处理标题一样：选择要填充的元素，然后设置它的内容。这个方法用于简单视图还好，但如果视图展示了大量数据，通过写代码来填充每个独立元素就会是一项单调且容易出错的工作。

另一种方法是在数据模型中的数据与视图标记中的元素之间创建一种自动映射。这个过程称作数据绑定（data binding）。这里有两种方式。一种是单向数据绑定（one-way data binding，见下图），数据被自动插入到标记中。这种方式对只读的表单和其他不可编辑的元素有效。



第二种是双向数据绑定（two-way data binding），数据可以被插入到标记中并且可以被读出来写入模型，特别是当表示数据已改变的事件被触发时。这种方式通常用于用户可编辑的数据。对于题目数据，我们需要一个可以用 HTML5 数据属性[3]创建的单向数据绑定的方案。

可以为视图的元素添加数据属性，来指定哪些属性应该绑定到这些元素上。这样的话，写一个函数把 JavaScript 对象的属性应用到一个元素上就是很容易的事了。我们需要把新元素和它们的数据属性添加到视图中：

earnjs/3200/public/index.html

```
<div class='templates'>
  <div class='problem-view'>
    <h3 class='title'></h3>
    <p data-name='description'></p>
    <pre><code data-name='code'></code></pre>
```

3 https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Using_data_attributes

```
  </div>
</div>
```

每个 data-name 指定应该绑定到该元素上的属性名。一旦把这些元素添加到标记中，就可以编写一个名为 applyObject 的数据绑定元素函数，它读取一个 JavaScript 对象，然后通过基于这个对象的字段名更新一个 jQuery 对象中的元素。使用 jQuery 的属性值 CSS 选择器语法，在选择特定属性值的元素时会很方便。在下面的代码中可以看到如何使用字段名构建 CSS 选择器表达式：

learnjs/3200/public/app.js

```
learnjs.applyObject = function(obj, elem) {
  for (var key in obj) {
    elem.find('[data-name="' + key + '"]').text(obj[key]);
  }
};
```

在题目视图中使用这个选择器表达式将题目数据绑定到视图元素中。

learnjs/3200/public/app.js

```
learnjs.problemView = function(data) {
  var problemNumber = parseInt(data, 10);
  var view = $('.templates .problem-view').clone();
  view.find('.title').text('Problem #' + problemNumber);
➤ learnjs.applyObject(learnjs.problems[problemNumber - 1], view);
  return view;
}
```

Joe 问：

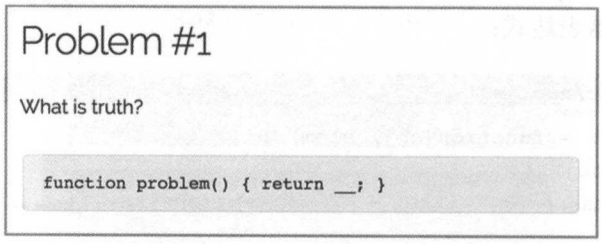
为什么不用$.data()?

你也许觉得奇怪，为什么不在 applyObject 中使用 jQuery 的 data 函数来读取 data-name 属性呢？原因是 jQuery 的 data 函数确实允许访问 HTML5 data 属性，但是除了该元素的 data 属性之外，它还维护着自己的 data 对象[4]，我不想混淆两者。

4 https://api.jquery.com/data/#data-html5

现在我们有一个可复用的单向数据绑定函数。当然，它不是万能的，它只能对文本值的元素起作用，对<input>、<select>或者<textarea>等表单元素无效。但是现在我们还不需要对这些元素做单向数据绑定。我们也许永远不会用到它，如果需要，也可以到时候再添加，跟现在添加一样容易。就当下而言，这段代码要尽可能简单。

我们来看看自己的成果。如果导航到题目视图，你应该可以看到如下图所示的界面。



如果手动把 URL 改成#problem-2，应用应该会切换到下一题。如果这些都能正常运转，就可以进行下一步开发了。

优化数据模型

本章开头，我们创建了一个简单的数据模型——只包含一个数组，数组里面有一些对象。随着应用功能的完善，这个数据模型也会变得复杂。我们需要增加一种新的数据类型，并将这种数据持久化到数据库。当你开发自己的无服应用时，可能需要更多数据类型来表达数据。

与其创建带行为的对象，我们更应该考虑在数据结构中保存数据，这样就能更方便地调用浏览器内建的 JavaScript 函数来操纵它们。然后创建简单的独立函数，将它们和数据关联在一起来构造复杂的行为。

比如，创建一个叫作 formatCode 的函数，它对题目对象中的代码应用格式化器。为了新建一组带格式的代码（formatted code）的答案对象，可以遍历这个列表，然后统一格式化。

learnjs/3913/public/tests/format_spec.js

```
var formattedProblems = [];
learnjs.problems.forEach(function(problem) {
  formattedProblems.push({
    code: learnjs.formatCode(problem.code),
    name: problem.name
  });
});
return formattedProblems;
```

如果你熟悉 JavaScript 中的 Array.map，就会发现这段代码可以轻松替换。只需要改一下 formatCode，将其改成接收一个题目对象而非纯文本作为参数，然后这段代码就可以简化成：

learnjs/3913/public/tests/format_spec.js

```
return learnjs.problems.map(learnjs.formatCode);
```

formatCode 接收一个对象作为参数，返回一个新对象，而不是接收文本参数后返回文本，因而很容易与其他函数组合。如果你想要一组代码排列整齐，并且根据名称排过序的答案对象，可以创建一个 byName 函数来作排序比较，然后调用它。

```
learnjs.problems.map(formatCode).sort(byName);
```

创建一个对同一类对象进行操作的函数库来代替对象层级，我们就能开发出一个丰富且有表现力的数据模型，而且此模型符合 Web 标准，使数据映射十分简单。如果我们在后面决定把题目集移到数据库里而不是直接保存在应用中，则不需要改变应用的任何逻辑，因为这个对象没有变。

我们要避免的另一个挑战（或者使其尽可能轻松）是数据迁移（data migration）。这个过程中大部分的艰难之处都可以通过把静态数据保存到应用里来缓解，我们对题目集就是这样处理的。然后就可以将这些变化部署到数据模式（data schema）上，和应用逻辑一起回滚。

举个例子，因为用户不能修改题目列表，我们可以把它存储在应用里。如果希望能够对这个题目列表重新排序，而且 ID 保持一致，就用有规律的主键来代替作为天然关键字的

数组序号。我们通过在这个对象中引入新的字段来实现，使应用能根据 ID 搜索题目，然后一键把所有代码推到 S3 上来部署这些修改。

考虑函数式的库

为数据模型开发领域专有的（domain-specific）函数库的同时，考虑一下引入一个库到应用，来支持函数式风格，如 Underscore.js[5]或 Ramda[6]。这些库提供许多标准 JavaScript `Object` 和 `Array` 类没有实现的函数，包括 `pluck`、`groupBy` 和 `union`。你甚至可以做真正的函数组合，也可以创建偏函数（partially applied function）。如果采用这种数据建模方法，使用函数式的库对你的应用来说是一个非常有价值的助力。

在开发自己的无服单页应用时，你可能会发现让数据模型尽可能简单和灵活是非常有益的。不要认为必须把数据保存到数据库——至少现在不用。可以稍后再把它们移到数据库，这样你的数据模型不会被数据以外的东西污染。JavaScript 提供的简单数据结构很强大，足以为不同类型的问题建模，而且便于用简便的工具进行操作。

避开难题通常是解决难题最简单的方法。当然，不可能总是完美地避开它们。一旦开始把数据存储到数据库，数据迁移就会变得有些棘手。我们将在第 5 章讲解这个过程。

既然我们已经定义了一个灵活的数据模型，那就用起来吧。在下一节，我们会让数据在屏幕上显示出来以便用户能看到它们，然后再讲解如何为用户提供视觉反馈，以及如何在不同的视图之间控制导航。

处理用户输入

`<form>`元素的本意是让网页能将用户提交的数据发回服务器[7]。表单的 `action` 属性

5 http://underscorejs.org/

6 http://ramdajs.com/

7 https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Forms/Sending_and_retrieving_form_data

和 method 属性允许你指定，当用户单击表单中的“提交”按钮时如何提交数据以及提交到哪里。不过我们想继续使用表单那种熟悉的外观和风格，因此需要找到一种方法捕获用户输入而不提交请求到服务器。我们希望应用阻止表单的提交而且完全在浏览器里执行。

既然应用把题目呈现给了用户，我们就要为用户提供一个提交答案的方式。我们希望用户输入他们的答案，单击按钮表示答题完成，然后在某个地方看到结果。应该告诉用户他们的答案是否正确，但是不应该提供任何其他信息（不能有提示！）。我们打算使用 HTML 的<form>元素来构造这个界面，因此必须拦截所有会引发浏览器发送数据的提交和点击事件，然后在应用里自行处理这些数据。

我们需要添加一个<textarea>到视图里来实现这部分工作流。这个元素将为用户提供足够的空间输入答案代码，毕竟这些代码有时候可能有几十行。我们还需要一个按钮供用户单击来提交答案，以及一些反馈表单来告知用户答案是否正确。

为了测试这个行为，必须模拟用户输入。一种方法是设置<textarea>的值，然后用 jQuery 触发按钮的点击事件。把这些测试添加到题目视图的 describe 部分，你就可以在 Jasmine 的 beforeEach 函数里初始化时候在作用域里创建一个可复用的 view 变量。下面这两个测试复用了外部作用域的那个 view 变量：

learnjs/3300/public/tests/app_spec.js

```
describe('answer section', function() {
  it('can check a correct answer by hitting a button', function() {
    view.find('.answer').val('true');
    view.find('.check-btn').click();
    expect(view.find('.result').text()).toEqual('Correct!');
  });
  it('rejects an incorrect answer', function() {
    view.find('.answer').val('false');
    view.find('.check-btn').click();
    expect(view.find('.result').text()).toEqual('Incorrect!');
  });
});
```

为了通过这些测试，必须让应用在用户点击按钮时做出响应。和着陆页的启动按钮不

同，你需要编程处理这个点击，所以要用 jQuery 添加一个点击事件处理函数。点击事件处理函数返回 `false`，告诉浏览器不要提交表单。提交表单会引起页面刷新，应用的状态将被重置，而这正是我们想避免的。

learnjs/3300/public/app.js

```
line 1 learnjs.problemView = function(data) {
  - var problemNumber = parseInt(data, 10);
  - var view = $('.templates .problem-view').clone();
  - var problemData = learnjs.problems[problemNumber - 1];
  5 var resultFlash = view.find('.result');
  -
  - function checkAnswer() {
  - var answer = view.find('.answer').val();
  - var test = problemData.code.replace('__', answer) + '; problem();';
 10 return eval(test);
  - }
  -
  - function checkAnswerClick() {
  - if (checkAnswer()) {
 15 resultFlash.text('Correct!');
  - } else {
  - resultFlash.text('Incorrect!');
  - }
  - return false;
 20 }
  -
  - view.find('.check-btn').click(checkAnswerClick);
  - view.find('.title').text('Problem #' + problemNumber);
  - learnjs.applyObject(problemData, view);
 25 return view;
  - }
```

Joe 问：

eval()是一个邪恶的函数吗？

是的。除了用户输入非法的 JavaScript 会抛一个 uncaught 的错误，像这样使用 `eval()` 会让我们的应用很容易遭受跨站脚本（XSS）攻击（参见第 7 章）。例如，如果有人诱导

用户随意输入 JavaScript 作为题目的答案，什么糟糕的事情都可能发生。后面我们会讲解一种在“沙盒”里安全地评估 JavaScript 的方法，这个“沙盒”与应用的其他部分是隔离的。这样的话，风险就很容易控制了。

除了点击事件处理函数，第 7 行的 `checkAnswer()` 函数能验证用户答案的有效性并判断它是否正确。这个函数构造一个 JavaScript 字符串检查是否有真值返回，然后对它调用 `eval()` 来得到结果。

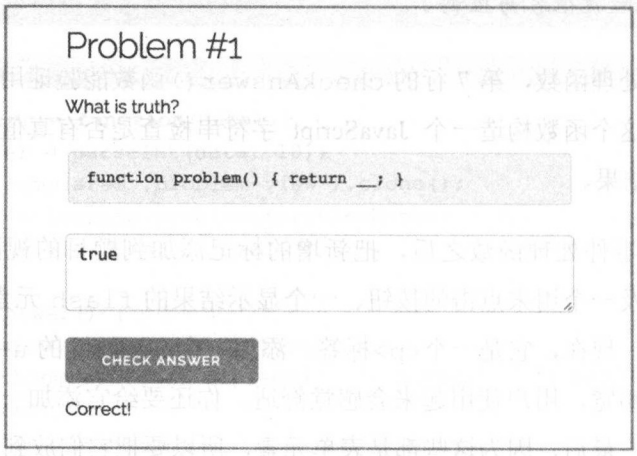
创建这个点击事件处理函数之后，把新增的标记添加到题目的视图模板里。要用到 `<textarea>`，以及一个用来点击的按钮、一个显示结果的 `flash` 元素。这个 `flash` 元素可以是任何东西。现在，它是一个`<p>`标签。添加一个 Skeleton 的 `u-full-width` 类，让`<textarea>`足够宽，用户使用起来会感觉舒适。你还要给它添加一个类，使它的字体对于代码更加友好。最后，因为这些都是表单元素，所以要把它们放到一个`<form>`里。

learnjs/3300/public/index.html

```
<div class='templates'>
  <div class='problem-view'>
    <h3 class='title'></h3>
    <p data-name='description'></p>
    <pre><code data-name='code'></code></pre>
➤   <form>
➤     <textarea class='u-full-width answer'></textarea>
➤     <div>
➤       <button class='button-primary check-btn'>Check Answer</button>
➤       <p class='result'></p>
➤     </div>
➤   </form>
  </div>
</div>
```

如果添加这个标记之后测试陷入无限刷新的死循环，检查一下你是否在按钮的点击事件处理函数底部添加了 `return false` 语句。这里的情况是，按钮被点击后会提交表单，使页面刷新，页面刷新就会执行测试，而执行测试就又会提交表单……

如果你的测试没有陷入无限刷新的死循环，很可能就通过了测试！好棒！如果通过了测试，切换回运行应用的浏览器标签看一看，如下图所示。



一切正常。虽然有点儿简单，但是它能正常运行。在花时间优化布局之前，我们希望先保证工作流是正确的，这个视图的工作流中还有一些需要我们注意。如果输入一个错误的答案（如 `false`）之后再输入另一个错误答案（如 `1==2`），这个视图根本没有变化。接下来我们将解决这个问题。

有效地使用视觉反馈

在传统 Web 应用中，用户的操作往往伴随着页面的跳转。虽然这不是理想的用户体验，但是用户对此很熟悉，而且它提供了一点重要的反馈：即让用户知道他们执行了某种操作，而应用对这项操作进行了响应。为用户提供这种反馈是一个实用的单页应用必须实现的功能。

题目视图现在就缺少这种反馈。无论我们的解决方案是否正确，如果用户输入的答案实际上没有变化，就不会有任何反馈告诉用户应用已经响应并检查了答案。假设有人认为自己知道正确答案，但是他输入的时候有一处笔误。这个用户可能会一遍又一遍地点击“检查答案”按钮，虽然应用每次都检查了答案，但他会认为这个应用有问题。提供可视的反

馈可以解决这个问题。下面我们用 jQuery 的 Effects API 来实现。

jQuery Effects API 有一系列不同的产生特效的函数[8]。这些函数在 Web 开发中经常被滥用。不必要的闪烁、移动、展开和缩小元素会使用户疑惑，令人厌恶。用这些特效巧妙地提供反馈，会让用户确认应用一直在如他们所预期的那样运行。

在这里，我们将使用 `fadeOut` 和 `fadeIn` 特效让用户知道应用什么时候重新检查了他们的答案。我们可以在应用命名空间下创建一个名为 `flashElement` 的新函数。

learnjs/3400/public/app.js

```
learnjs.flashElement = function(elem, content) {
  elem.fadeOut('fast', function() {
    elem.html(content);
    elem.fadeIn();
  });
}
```

我们可以在题目视图里用这个函数，但是也可以在其他需要可视化反馈的视图里复用它，就像下面这样：

learnjs/3400/public/app.js

```
function checkAnswerClick(){
  if (checkAnswer()) {
    learnjs.flashElement(resultFlash, 'Correct!');
  } else {
    learnjs.flashElement(resultFlash, 'Incorrect!');
  }
  return false;
}
```

除了指定动画运行速度的`'fast'`字符串之外，`fadeOut` 函数在特效结束的时候会调用一个回调函数。你可以用这个回调函数来触发内容的变化并渐显出来。从用户的角度来看，这个视图特效就是已有的内容逐渐隐去，然后新内容逐渐显现出来，这样他们就知道自己的答案已经被理解了。

8 http://api.jquery.com/category/effects/

现在如果你加载应用并输入一个答案，每次我们检查这个答案的时候，你应该看到快速但清晰可见的切换。这就相当于向用户保证，即使他们看到的结果状态没有改变，但应用其实仍然在如期运行。

控制导航

传统 Web 应用常常联合使用视图和控制器来控制导航。一般来说，根据应用的状态有以下实现方法：直接把链接渲染到视图中；按需添加重定向到其他路由的路由；或者添加渲染不同视图的控制器。在我们的应用中，仅使用一个函数来实现。我们没有确切的控制器，所以控制导航的职责就交给视图函数。当渲染一个视图的时候，视图函数需要添加到其他视图的链接，或者添加能触发浏览器导航到其他地方的行为。

目前，我们的应用允许用户查看并解答第一道题。这很不错，但是我们需要通过一种紧凑方式引导他们接着解答下一题。为了实现这个目的，我们打算优化 problemView 函数来构造导航到下一题的链接。这个链接只有回答了当前题目以后才会显示出来。

为了创建这个链接，我们先创建一个模板。到目前为止，我们只对视图使用了模板，但是我们也能对需要创建的标记的任何片段使用模板。比如，添加一个 到 templates 中：

learnjs/3500/public/index.html

```
<div class='correct-flash'>
  <span>Correct!</span> <a>Next Problem</a>
</div>
```

既然有一个我们想要的模板了，接下来就复制它。这里重复写代码没什么意思，所以从 problemView 函数提取一个新函数作为开始吧。

learnjs/3500/public/app.js

```
learnjs.template = function(name) {
  return $('.templates .' + name).clone();
}
```

只要是创建模板，用这个模板函数都很方便，无论你是要用于视图还是用于像链接那

样的标记片段。最后，需要修改“Check Answer”（检查答案）按钮的点击处理函数。复制模板，然后更改 href 属性，使其指向下一题。

learnjs/3500/public/app.js

```
function checkAnswerClick() {
  if (checkAnswer()) {
    var correctFlash = learnjs.template('correct-flash');
    correctFlash.find('a').attr('href', '#problem-' + (problemNumber + 1));
    learnjs.flashElement(resultFlash, correctFlash);
  } else {
    learnjs.flashElement(resultFlash, 'Incorrect!');
  }
  return false;
}
```

像这样动态创建链接是我们的视图函数控制用户在应用里导航的一种方式。如果你想让应用来决定当点击按钮时导航到哪里，当然也可以。在 checkAnswerClick 点击处理函数中设置 window.location.hash 属性，我们能通过 JavaScript 控制逻辑把用户导航到任何视图。例如，对于错误的答案，如果想根据用户提供的答案的类型来显示不同的帮助链接，也可以在这个点击处理函数中轻松实现。

使用新 template 函数，能扩大模板的使用范围来创建 HTML 组件函数。这些函数封装了模板的初始化和创建，是一种让行为更加易于测试以及在视图间共享组件的好方法。举个例子，为了处理题目列表的最后一题，我们可能想将用户导航到其他地方。把他们送回着陆页似乎是一个不错的办法，就这么办！在为这个行为编写测试脚本之前，从答案按钮的点击处理函数抽取一个名为 buildCurrentFlash 的函数，然后直接测试这个函数。做完这些，你应该会得到如下代码：

learnjs/3700/public/app.js

```
learnjs.buildCorrectFlash = function (problemNum) {
  var correctFlash = learnjs.template('correct-flash');
  var link = correctFlash.find('a');
  (problemNum < learnjs.problems.length) {
    link.attr('href', '#problem-' + (problemNum + 1));
  } else {
```

```
    link.attr('href', '');
    link.text("You're Finished!");
  }
  return correctFlash;
}
```

把这个行为拖出来放到一个独立的函数中，不仅使它更易于测试，而且使视图行为更易于测试，因为它构造了一个易于用 spy 模拟的测试线缝。既然用户可以在应用里导航，接下来可以着手实现单页应用的另一个必要部分：应用外壳（shell）。

创建一个应用外壳

使用我们的应用时，你可能注意到重载题目视图有时会导致着陆页标记在屏幕上一闪而过。在触发加载应用的事件之前，浏览器渲染一次这个页面就会引起标记的闪烁。这种情况是否会出现，取决于应用的各种资源的加载速度。

这个标记闪烁是一种骚扰，也会使用户感到困惑。显然，这个问题需要解决。好消息是解决办法相当简单，而且这个改动将在应用里创建一个重要的新结构：外壳。

外壳是模板和视图容器之外的可视标记。外壳里的所有内容在每个视图里都是可见的。使用外壳可以为应用添加如导航条、Logo、侧边栏或者菜单等部件，我们也可以用它作为任何视图可见的固定位置元素的父元素，如聊天窗口、工具栏或者对话框。

提取着陆页

创建外壳，第一步是把 `landing-view<div>` 从 `view-container` 中移到 `templates <div>`。

learnjs/3800/public/index.html

```
<div class='markup'>
  <div class='view-container container'>
  </div>
```

```
<div class='templates'>
  <div class='landing-view'>
  <!-- Markup Continues... -->
```

这一步应该会无法通过测试。为了通过测试，需要为载入的视图创建一个视图函数。

learnjs/3800/public/app.js

```
learnjs.landingView = function() {
  return learnjs.template('landing-view');
}
```

然后将视图函数添加到 routes 对象中。

learnjs/3800/public/app.js

```
var routes = {
  '#problem': learnjs.problemView,
  '': learnjs.landingView
};
```

经过这些修改以后，测试应该会通过。创建一个应用外壳也修复了另一个问题：“返回”按钮正常了！如果你单击了“启动”按钮，然后在浏览器中单击“返回”按钮，你将回到着陆页。如果你想创建一个别名路由映射到这个视图，如"#landing"或者简单的"#"，也是可以的。为 routes 对象添加另一个入口，把该路由映射到 learnjs.landingView 函数即可。

添加工具条

现在我们已经把所有内容都提取出来加到视图里并且创建了一个应用外壳，下面就往这个外壳里添一些东西吧。给应用加上一个简单的工具条应该不错。在视图容器之上给标记添加一个 元素，此元素中有一个无序列表。

learnjs/3901/public/index.html

```
<div class='nav-container no-select fixed-top u-full-width'>
  <ul class='inline-list hover-links nav-list six columns'>
    <li><a class='text-lg' href="#">LearnJS</a></li>
    <li><a href="#problem-1">Start</a></li>
```

```
  </ul>
</div>
<div class='view-container container'>
</div>
```

如果好好装饰这个无序列表，它就可以成为我们的导航条。正如本例所示，你可以添加带有锚点（anchor）的列表元素，来创建到应用的链接。这里我们已经创建了两个映射到路由的链接：一个返回着陆页，一个跳转到题目视图。

为了装饰这些元素，我们打算添加一些类。Skeleton CSS 样板提供了 u-full-width 类，它确保导航条占满页面宽度。six 和 columns 类使链接列表位于导航条的左半部（12 列中的前 6 列），为右侧其他元素腾出了空间。

至于那些应用自定义规则的 CSS 类，我们希望根据它们的风格来组织和命名，而不使用对标记结构的描述来命名。例如，要显示我们的内联列表，就可以用 inline-list 类。

learnjs/3901/public/index.html

```
.inline-list {
  margin-bottom: 0px; /* Skeleton reset */
}
.inline-list li {
  display: inline;
  margin: 0 20px 0 0;
}
```

我们希望这个导航条固定在屏幕上方，即使用户把页面滚动到底部，导航条也是可见的。为了实现这种效果，可以为 fixed-top 类添加一条规则，将导航条定位在页面顶部，同时让它的 z-index 足够大使其位于其他元素之上。你还需要把 body CSS 规则中的 margin-top 属性从 30px 改成 60px 来适配导航条的新高度。

learnjs/3901/public/index.html

```
.fixed-top {
  position: fixed;
  top: 0px;
  z-index: 1024;
}
```

可以用 user-select 属性来防止用户故意选择导航条元素而不点击它们。目前不是所有的浏览器都支持这个属性，所以除了标准 CSS 属性名，你还需要用浏览器引擎前缀（vendor prefix）。

learnjs/3901/public/index.html

```
.no-select {
  user-select: none;
  -webkit-user-select: none;
  -ms-user-select: none;
  -moz-user-select: none;
}
```

浏览器引擎前缀

浏览器引擎前缀（vendor prefix）是一种惯例，在一个 Web 标准被批准和采纳之前，浏览器厂商通过它为浏览器添加一些实验性 CSS 功能。虽然很多浏览器支持 user-select CSS 属性，但它还不是万维网委员会（W3C）CSS 规范的一部分，所以使用多个带合适浏览器引擎前缀的属性可确保我们的应用能跨浏览器正常工作。

虽然我们在导航条中使用了链接，但不希望它们像普通链接一样有下画线。当用户在链接上悬停时我们会应用一个样式。可以用阴影或者其他 CSS 特效，但是对于这个应用，我们只是简单地加了一条下画线。为 hover-links 类创建一条规则，就能实现这样的效果：

learnjs/3901/public/index.html

```
.hover-links a { text-decoration: none; }
.hover-links a:hover { text-decoration: underline; }
```

我们需要设置工具条的颜色、尺寸和内边距，这样它就像正常的工具栏那样占满整个屏幕的宽度。和前面一样，还是通过为 nav-container 类创建规则来实现：

learnjs/3901/public/index.html

```
.nav-container {
  padding-left: 40px;
  background: #666;
}
```

```
.nav-container a { color: white; }
```

最后，不管你是否已经为着陆页的视图添加了别名路由，给着陆页视图添加一个，就像在第一个导航条链接中用的那个一样。

learnjs/3901/public/app.js

```
var routes = {
  '#problem': learnjs.problemView,
  '#': learnjs.landingView,
  '': learnjs.landingView
};
```

一旦添加了这些规则和元素，导航条就应该看起来像下图这个样子。

有了一个应用外壳和工具条，我们的应用就不再仅仅是一堆视图了。接下来，我们需要想办法把应有的所有部件组合起来。

使用自定义事件

现在，我们的视图可以轻松地唤起应用其他部分的行为。例如，数据模型函数能在视图之间方便地共享。但是有一件事被漏掉了，即应用在视图里触发行为的方法。目前还没有什么好办法能让应用在视图函数的作用域使事情发生。

我们能采取的一个办法是添加一个按钮到新工具栏中，然后在创建每个视图的时候注

册一个点击处理函数。然而要是没有一种机制能删除这个点击处理函数，这个作用域里的处理函数和变量将永远不会被垃圾回收。如果用户在视图中跳转，应用的内存使用率会越来越高，直到应用崩溃。

语义化 CSS 类

在本例中我没有解释各个独立 CSS 属性的含义，这些东西在网上都能找到。不过，你可能希望能够从类名称中看出每个规则背后的含义。我喜欢把规则按语义命名，而不是把一个类名放到一个元素里，然后用这个类来装饰它。因为按语义命名不仅使代码更容易理解，也使得它更有可能被复用。

为了避免这个问题，我们可以使用自定义事件（custom event）来给视图发送消息。如果还不确定当前视图是否需要对应用的其他行为做出响应，这种方法就很有用，例如在视图被替换时删除事件处理函数。通过给视图注册一个事件监听器，我们能触发视图里的行为而不必打破视图函数的封装，也不会造成内存泄漏使应用不稳定。

添加这些事件时，要确定谁会触发它们。在这里，路由器知道它会在什么时候用另一个视图替代当前视图。有时视图在被替换时会抓着本应释放的资源不放，事件订阅给了该视图这样做的理由。

我们没有用类库或者框架来管理这些事件，而是依据 Web 标准进行开发。我们准备在顶层针对应用创建一些函数。你应该会觉得这个方法有点似曾相识。

具体来说，就是使用文档对象模型（DOM）已有的事件系统，触发视图能监听到的自定义事件。DOM 事件在元素上被触发，顺着元素层级冒泡。我们能通过在 `view-container` 元素（只能是视图）的任何子元素上触发事件来给视图发送事件。视图可以通过在元素上绑定事件监听器来订阅这些事件。为了触发这些事件，可以在 `learnjs` 命名空间里创建一个新函数：

learnjs/3912/public/app.js

```
learnjs.triggerEvent = function(name, args) {
```

```
  $('.view-container>*').trigger(name, args);
}
```

利用新的事件机制，我们可以为题目视图在导航条上添加一个“Skip”（跳过）按钮。这个按钮允许用户跳过当前题目，即使他们还没回答完这一题。这个按钮只有在题目视图加载之后才显示，当题目视图被移除的时候按钮也随之消失。

为了在一个视图被移除时你还能进行操作，必须在这时触发一个事件。这就需要对路由函数做一些修改。在该视图被替换之前，可以触发事件让所有现存的视图知道它们将要被移除。当新的视图被创建之后，你可以在 showView() 函数里用如下代码实现：

learnjs/3912/public/app.js

```
learnjs.triggerEvent('removingView', []);
$('.view-container').empty().append(viewFn(hashParts[1]));
```

现在，只需要修改题目视图来添加和删除按钮了。首先，需要为 problemView() 函数添加必要的行为。一旦为 skip-btn（接下来要实现）添加必要的标记，下面的代码应该就能用了。别忘了检查我们现在是否位于题目列表的末尾。同时，注意 jQuery 的 bind 函数的使用会给视图元素添加一个事件监听器。

learnjs/3912/public/app.js

```
if (problemNumber < learnjs.problems.length) {
  var buttonItem = learnjs.template('skip-btn');
  buttonItem.find('a').attr('href', '#problem-' + (problemNumber + 1));
  $('.nav-list').append(buttonItem);
  view.bind('removingView', function() {
    buttonItem.remove();
  });
}
```

接下来，我们需要对标记做几处改动。首先，为“Skip”按钮创建一个模板。你可以把它添加到之前创建的 templates 里。

learnjs/3912/public/index.html

```
<li class='skip-btn'>
  <a>Skip This Problem</a>
```

```
</li>
```

做完这些修改之后，这个“Skip”按钮应该能正常工作了。如果你导航到题目视图再返回着陆页，就会发现按钮不见了。用 jQuery 的 `empty` 函数把题目视图从 DOM 里移除，清除关联数据和事件处理函数，这样就不用担心要如何清除清理函数了。

现在这个事件系统已经就绪，它能做很多不同的事情。我们开始访问 Web 服务时，就可以用事件通知视图，模型里的数据已经更新，或者用户已经完成登录或者退出等行为。我们可以愉快地用它来清除引用，防止内存泄漏。是时候部署应用的新版本了。

再次部署

现在我们的应用已经具备一些功能，可以再部署一次了。用户现在能查看和解答 JavaScript 题目，而且能从一道题导航到下一道题。在本章中，我们只添加了一小部分题目到列表。如果你愿意，还可以添加更多。从简单的功能开始实现，然后慢慢添加新功能，让应用变得有趣！

下一步

本章我们学习了如何快速、迭代地开发 Web 接口，下面这些话题你可能会感兴趣。

Web 无障碍访问

不是所有用户都使用的是主流浏览器。视障用户会使用屏幕阅读器，它提供一种完全不同体验。很多第一次使用屏幕阅读器的 Web 开发者会发现其使用体验简直糟透了。

设计无障碍访问的 Web 应用是一个很深的话题，超出了本书的讨论范围，但是我可以帮助你了解一些基础知识。下面是一些关于创建无障碍访问应用的有用资源：

http://www.w3.org/standards/webdesign/accessibility

http://itstiredinhere.com/accessibility

创建桌面图标

很多移动设备支持添加到 Web 应用的链接作为桌面按钮。这些链接和原生应用出现在一起，你甚至可以指定一个图标来代表它们。可惜的是要实现这些效果需要使用厂商特有的技术。不过对于大部分厂商而言，添加一个必要的 meta 元素到应用是连接用户的好方法。

iOS：https://developer.apple.com/library/ios/documentation/AppleApplications/Reference/SafariWebContent/ConfiguringWebApplications/ConfiguringWebApplications.html

Chrome for Android：https://developer.chrome.com/multidevice/android/installtohomescreen

CSS 动画

我们提到了如何用 jQuery 特效来实现视觉反馈，但它们不是唯一的方案，甚至连最常用的都算不上。CSS 3 动画[9]也能实现同样的特效。有些特性浏览器不见得会支持，不过 CSS 动画是个例外。

表单验证

前面我们在题目视图里检查答案的流程叫作表单验证（form validation）。不过这个应用里的验证是专属的，你可以找到一些执行更常规的表单验证的库，例如 validate.js 和 Parse.js。

下一章，我们将为应用添加很多 Web 服务。我们会介绍用户身份验证，并且把应用连接到其他厂商提供的社交媒体账号上，例如 Google 和 Facebook。我们自己不开发系统来存储账号、密码和其他敏感的用户数据，而是让 Amazon 来做这些事。

---

9 https://developer.mozilla.org/en-US/docs/Web/CSS/animation

第 4 章
基于Amazon Cognito的认证服务

身份认证是大多数应用的必要组成部分。所有收集用户数据的软件都必须采用某种方法组织、访问和保护这些数据。在一个传统的 Web 应用中，应用服务器经常通过浏览器 cookie 管理认证。但是由于浏览器实施的一些规则，不同来源[1]的 cookie 不能共享。这意味着如果你的应用服务器管理使用 cookie 的认证令牌，为了访问这个 cookie 和认证请求，你的其他所有 Web 服务必须共享一个源。

不过，如果把身份认证管理仅仅当作一项 Web 服务，这些顾虑大多都能化解。在应用层（JavaScript 内）管理的安全证书不需要担心同源策略。这意味着你不需要任何中间件存放所有的密钥或作为代理就可以在浏览器里直接使用大量服务。

在无服应用中，你可以把身份认证管理的责任交给第三方 Web 服务，让它来处理所有难题和风险。如果有一个安全的 Web 服务能管理身份认证，并且为其他 Web 服务提供认证功能，将彻底改变我们开发 Web 应用的方式。Amazon 的 Cognito 就是一个这样的身份管理服务。

Cognito 服务允许我们通过身份联盟（identity federation）管理身份认证。这代表我们

1 https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

能使用多个身份认证服务商（例如 Google、Facebook，甚至我们自己的应用）的身份，并且把它们关联到 Cognito 提供的一条身份记录上。Cognito 把这些记录保存在身份池（identity pool）中。Amazon 的 IAM 控制台允许我们定义策略来授权身份池里的用户访问所有 Amazon 不同级别的 Web 服务。我们的应用不需要将请求路由到任何应用服务器就能代表认证用户直接从浏览器发送请求。认证服务变成了基础设施的责任而不是应用的责任。

在传统 Web 应用中，像这样的身份信息通常存储在数据库中。你必须有一张用户表，而且可能会使用这张表中的记录的主键关联其他表的记录。如果你了解自己在做的事情，这条记录会包含妥善加密过的（salted and hashed）密码。如果你根本不知道“妥善加密”是什么意思，那么 Cognito 就是你的救星。这里我们不准备管理自己的用户记录，而是使用 Cognito 创建的身份标识，并且在所有写到数据库的记录里都包含它。使用了 Cognito，我们就不需要管理所有这些基础设施，并且避免了在管理用户密码时产生的安全隐患。

在本章，我们将学习如何管理身份认证而不需要应用服务器保存所有密钥。为了启用社交账号登录，我们会创建一个身份池作为用户身份的仓库。我们会接入第三方身份认证服务商，例如 Google 和 Facebook，然后创建允许我们的应用直接从浏览器访问所有 Amazon Web 服务的配置文件。之后，我们就可以开始下一步：将用户数据保存到数据库。

接入外部身份认证服务商

Google、Facebook 和其他服务商创建的身份认证访问系统是大体上基于 OAuth2 标准的，该标准允许网络应用和其他客户端获得通过 HTTP 访问信息的临时权限。虽然 OAuth2 标准本身还有一些严重的问题[2]，但是主流服务商所实现的身份认证访问系统在这一点上处理得很好，并且提供了很多集成的机会。

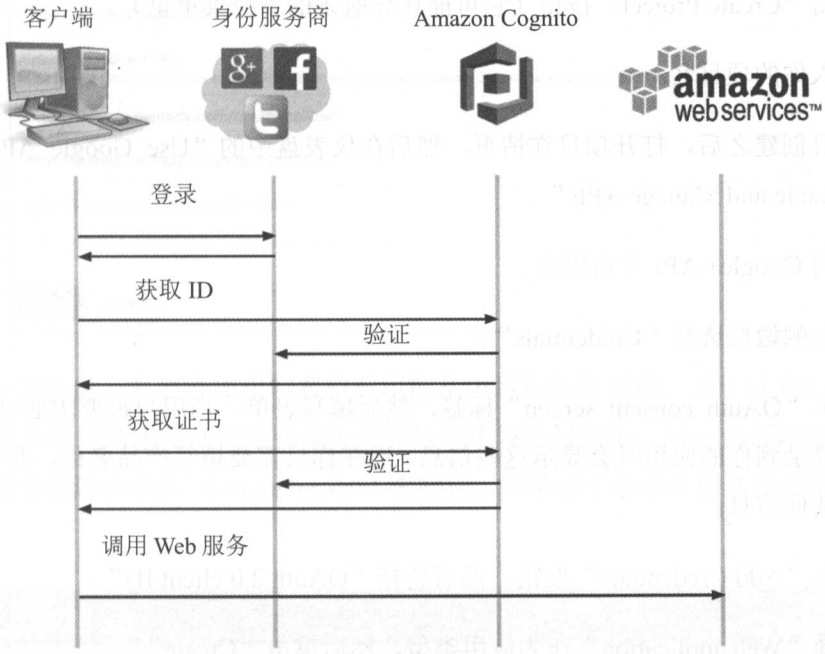
使用 Cognito 来为我们管理这些交互，意味着我们能充分利用 OAuth 的优点，同时还

2 http://hueniverse.com/2012/07/26/oauth-2-0-and-the-road-to-hell/

能避免它的一些问题。首先，我们需要从身份服务商获取唯一的身份标识。每个服务商都有一套自己的制作身份表示的方法。一旦获取了这个 ID，就可以和一个 Cognito 身份配对，然后用这个身份去获取 AWS 证书。

Cognito 在一些 AWS 服务区不可用。

把这些步骤串起来，就能看出用户是如何接入我们的应用的。首先，他们通过第三方身份服务商登录。在这个登录过程中需要在服务商的站点以及和一个确认页面上输入用户名和密码。整个过程如下图所示。



一旦登录，我们的应用就能获取服务商的 ID 令牌，然后把它和我们的身份池 ID 一起发送给 Cognito。Cognito 用这个令牌重新连接身份服务商来确认用户身份。然后把这个第三方身份与一个新的 Cognito 身份关联，并把这个 Cognito 身份添加到身份池。Cognito 会返回这个身份信息，我们在后面可以用它来请求临时 AWS 证书。如果第三方 ID 令牌合法，

Cognito 会自动刷新证书。我们的应用使用这些 AWS 证书向身份池中认证用户配置授权的服务发出 Web 服务请求。

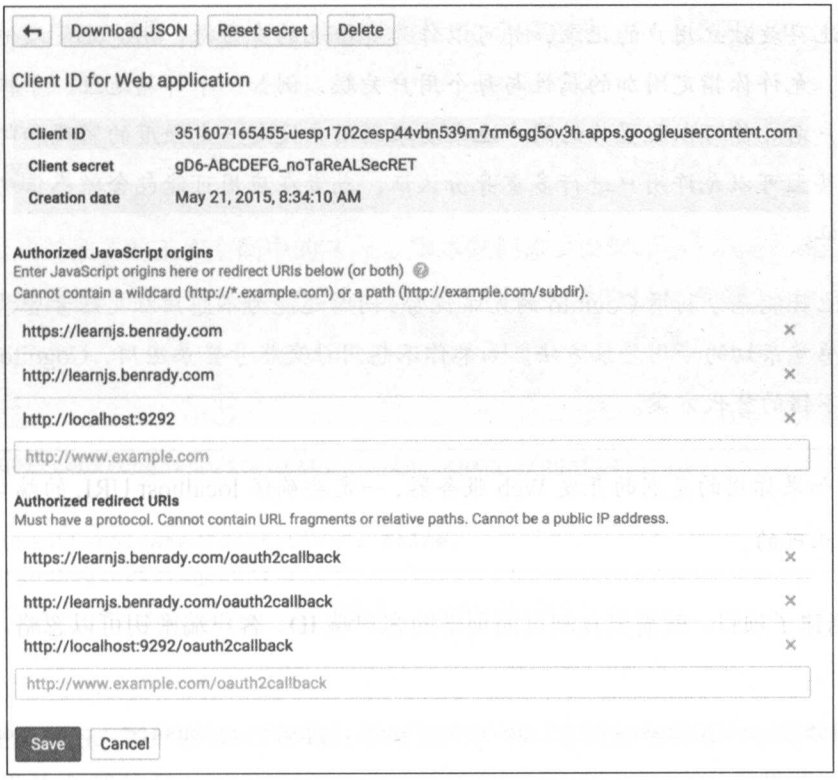
Google 是我们添加的首个第三方身份服务商，你也可以通过类似的流程添加其他服务商。为了获得发送给 Cognito 的 Google ID，必须遵循 Google 要求的 Google+ 登录流程（Google+ Sign-in）[3]。流程的第一步是在 Google 开发者控制台（也称作 Google API 控制台）中创建和配置一个 Google 项目。步骤如下：

1. 打开 Google 开发者控制台[4]，然后用你的 Google 账号登录。
2. 单击“Create Project”按钮（它可能在导航条的下拉菜单里）。
3. 输入你的项目名。
4. 项目创建之后，打开项目详情页，然后在仪表盘中的“Use Google APIs”下选择“Enable and Manage APIs”。
5. 找到 Google+ API 并启用它。
6. 在左侧边栏选择“Credentials”。
7. 点击“OAuth consent screen”标签，然后填写表单。当用户被要求通过 Google 账号登录到你的应用时会显示这些信息。现在你只需要填写产品名称，也可以随意增加其他信息。
8. 单击“Add Credentials”按钮，然后选择“OAuth 2.0 client ID”。
9. 选择“Web application”作为应用类型，然后单击“Create”。

完成之后你会看到如下图所示的表单。

3 https://developers.google.com/identity/sign-in/web/sign-in

4 http://console.developers.google.com



在这个界面里，需要添加我们希望应用能从中运行的所有源。当认证我们的应用时，Google 会认为认证请求是从在这些地方运行的应用发送过来的。这一点可以通过浏览器发送请求时添加的源请求头[5]（origin header）来验证。如果这个请求来自其他任何地方，将会被拒绝。

Cognito 用户池

除了像 Google 和 Facebook 这样的第三方认证服务商，Cognito 还提供一种为你的应用创建专属用户账号的机制。这些账号与第三方身份无关，需要用密码来认证。这类账号可以用一个 Cognito 用户池（user pool）来管理。

5 https://developer.mozilla.org/en-US/docs/Glossary/Origin

用户池存放独立用户的记录，并可以作为身份池的身份源。用户池管理他们自己的认证证书，允许你指定附加的属性与每个用户关联，例如一个邮箱地址、手机号码、生日或者用户名。你可以配置你的用户池来要求新用户设定一定强度的密码和邮箱或短信验证。你甚至可以允许用户进行多重身份认证，或者在应用可能包含敏感信息时激活多重身份认证。

用户池既能充分利用 Cognito 的无服优势，同时还能为不想用社交账号登录的用户提供让他们感觉亲切的专用登录方法。如果你不想用社交账号登录应用，Cognito 用户池可能是一个不错的替代方案。

如果你用的是别的开发 Web 服务器，一定要确保 localhost URL 的端口号是正确的。

一旦创建了项目，就需要复制页面顶部的客户端 ID。客户端密钥可以忽略。客户端 ID 是这个样子：

```
351607165455-uesp1702cesp44vbn539m7rm6gg5ov3h.apps.googleusercontent.com
```

既然已经在 Google 开发者控制台中创建了一个应用，就能用这个应用的客户端 ID 将应用和 Google 连接起来。下一步，我们要在 Cognito AWS 控制台中创建身份池。

创建身份池

身份池是存放我们应用的用户身份的容器。它有点像数据库中的一张 `User` 表，保存着用户的列表，并且显示他们曾使用过何种凭证进行连接。创建一个身份池是相当简单的事情，但是要理解其工作原理就要深入理解 Amazon IAM 了。

身份池里的身份可以是经过（第三方身份服务商）认证的，也可以是未被认证的，即它们本质上是匿名的。你可以在不同应用中复用身份池，这样就能在这些应用中共享你的

用户数据。Amazon 设置了限制，一次只能激活 60 个身份池，但是不限制单个身份池中的身份（用户）个数。

身份池配置

你可以通过预备的工作空间中的 sspa 脚本来创建身份池。create_pool 操作会根据你指定的配置目录创建一个新身份池，以及其中所有关联的 IAM 配置。在备好的工作空间里，打开 conf/cognito/identity_pools/learnjs 目录。打开 config.json 文件看一看：

learnjs/4001/conf/cognito/identity_pools/learnjs/config.json

```
{
   "AllowUnauthenticatedIdentities": false,
   "SupportedLoginProviders": {
      "accounts.google.com": "<Google Client ID Here!>"
   }
}
```

一旦把 Google 客户端 ID 添加到这个配置文件里，它就会包含 sspa 脚本创建一个新身份池所需的全部信息。AllowUnauthenticatedIdentities 属性指定池子里的 Cognito 用户是否必须用第三方身份服务商认证。不过我们不要求所有的用户必须登录才能使用应用，而是在他们想要保存自己的答案时要求他们登录。

SupporteLoginProviders 列出了我们所希望支持的身份服务商，以及 Cognito 用于验证这些服务商令牌的 ID。如果你想要添加别的身份服务商，可以在创建身份池之前把它们添加到列表里。如果准备就绪，执行下面的命令：

```
learnjs $ ./sspa create_pool conf/cognito/identity_pools/learnjs
```

sspa 脚本调用我们在第 1 章安装的 AWS CLI 工具和你指定目录下的配置文件来创建新身份池。身份池的名字会和配置目录一样。如果你想要创建另一个身份池，可以创建一个新目录，并添加一个 config.json 文件（也可以只是一个符号连接），然后再次运行这条命令。这对创建一个测试环境很有帮助。

运行命令之后，在配置目录里会出现一些新文件。逐一查看它们，就会理解 sspa 脚本做了些什么。

第一个被创建的文件是 pool_info.json。这个文件包含身份池被创建之后从 Cognito 服务返回的信息。正如你所见，除 config.json 文件包含的信息之外，你还会看到身份池的 ID。

learnjs/4001/conf/cognito/identity_pools/learnjs/pool_info.json

```
{
  "IdentityPoolId": "us-east-1:71958f90-67bf-4571-aa17-6e4c1dfcb67d",
  "AllowUnauthenticatedIdentities": false,
  "SupportedLoginProviders": {
    "accounts.google.com":
      "ABC123ADDYOURID.apps.googleusercontent.com"
  },
  "IdentityPoolName": "learnjs"
}
```

除了创建身份池本身，这个脚本还为我们的应用创建了一些其他资源。在讨论这些资源之前，我们需要花点时间理解用户如何通过身份池获得访问 Web 服务的权限。

IAM 角色和策略

使用 Cognito 身份池的一个原因是它允许用户简单地设定 IAM 角色。角色在很多方面和 IAM 用户很像，因为能通过策略指定角色什么操作能做而什么不能做。和用户不同的是，角色被用于很多不同的用户，用户可以设定角色来完成某些操作。为了让用户访问我们的应用需要的所有服务，必须给他们授权。一种做法是在身份池里为已认证的用户创建一个角色。

为了让用户设定一个角色，你必须为这个角色创建一个策略。创建角色时，需要包含这条策略，称为角色设定策略（assumed role policy）。注意，这条策略和其他所有授权用户访问服务（例如从 DynamoDB 表中读）的策略是分开的。sspa 脚本在一个被添加到身份池配置目录的名为 assume_role_policy.json 文件里定义这条策略。它应该是下面这样：

learnjs/4001/conf/cognito/identity_pools/learnjs/assume_role_policy.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Federated": "cognito-identity.amazonaws.com"
      },
      "Action": "sts:AssumeRoleWithWebIdentity",
      "Condition": {
        "StringEquals": {
          "cognito-identity.amazonaws.com:aud":
            "us-east-1:71958f90-67bf-4571-aa17-6e4c1dfcb67d"
        },
        "ForAnyValue:StringLike": {
          "cognito-identity.amazonaws.com:amr": "authenticated"
        }
      }
    }
  ]
}
```

第 1 章提到 IAM 策略，当时我们第一次安装 AWS CLI。你可以给用户或角色添加其他策略来授权访问服务。在第 1 章中我们使用的是一条托管策略（managed policy），名为 AdministratorAccess。我们刚才使用的策略是一条自定义策略（custom policy）。自定义策略能精确控制对特定资源的访问，给特定类型的用户访问权限，或者在一些条件下限制或开放访问权限。我们来看一看这个策略文档里的几个属性，看看它们做了什么。

这条策略只有一条语句（statement），其中的 `Effect` 条目表明它授权一个操作（而不是撤销它）。`Action` 段指定这条策略会授权使用 `sts:AssumeRoleWithWebIdentity` 操作。Cognito 用 Amazon 的安全令牌服务（Security Token Service，STS）为用户生成临时 AWS 证书。`AssumeRoleWithWebIdentity` 操作允许经第三方身份服务商认证的用户设定这个角色，然后获取这些生成的证书。

策略中的 `Condition` 条目要求当键的 hash 值与定义的值匹配时才执行声明中语句的

所有行为，以此来进行精细的访问控制。策略里的第一个 Condition 子句约定这个角色只被用于一个身份池，每个身份池都有自己的池 ID。Condition 里的另一个子句约定此角色仅限经认证的 Cognito 用户使用。如果你希望允许未经认证的用户访问 AWS 服务，可以通过他自己的角色设定策略创建一个单独的角色。

sspa 脚本用这个策略来创建一个新角色。你可以在 role_info.json 中看到 AWS 的响应。

learnjs/4001/conf/cognito/identity_pools/learnjs/role_info.json

```
{
  "Role": {
    "AssumeRolePolicyDocument": {
      "Version": "2012-10-17",
      "Statement": [
        { "Action": "sts:AssumeRoleWithWebIdentity",
        "Principal": {
          "Federated": "cognito-identity.amazonaws.com"
        },
        "Effect": "Allow",
        "Condition": {
            "StringEquals": {
              "cognito-identity.amazonaws.com:aud":
                "us-east-1:71958f90-67bf-4571-aa17-6e4c1dfcb67d"
            },
            "ForAnyValue:StringLike": {
              "cognito-identity.amazonaws.com:amr": "authenticated"
            }
          },
          "Sid": ""
        }
      ]
    },
    "RoleId": "AROAJ5QOAOJOWUAORXS3S",
    "CreateDate": "2015-12-11T02:23:13.981Z",
    "RoleName": "learnjs_cognito_authenticated",
    "Path": "/",
    "Arn": "arn:aws:iam::730171000947:role/learnjs_cognito_authenticated"
  }
}
```

除了我们讨论过的 AssumeRolePolicyDocument，在响应里还可以看到关于此角色的其他信息，包括名称、Amazon 资源名（Amazon Resource Name，ARN）和创建时间。

你可以在 Cognito AWS Web 控制台中查看新创建的身份池。

这样，身份池就创建好了。现在可以将 Cognito 认证整合到应用里。我们需要添加一个按钮到导航条，使用户能使用 Google 账号登录。我们还需要添加一个视图，让用户检查他们留存在应用中的信息。

获取 Google 身份

使用 Cognito 认证与使用第三方身份服务商的一个区别，是用户不用登录——至少不是传统意义上的登录。因为我们通过 Cognito 关联身份，用户可以通过第三方身份服务商连接到应用中，我们根本不需要识别和保存密码。因此，我们不需要登录页面，但是需要一个途径让用户通过他们想用的身份接入进来。现在，我们就继续使用 Google 身份，当然如果你想的话，也可以用同样的方式添加更多身份。

我们已经在 Google 开发者控制台中创建了一个应用，并且将其连接到我们的身份池。现在需要构建让用户接入应用的真实流程。正如你在前面所见，这个过程称为“Google+ Sign-in”。我们需要在导航条中添加一个“Sign-in”（登录）按钮，当用户想登录的时候有门可入。

你可能会觉得 Google 这个添加登录按钮的流程有点令人反感，因为需要让 Google 直接在我们的标记中插入这个按钮，而不是调用 API。好吧，那就加载和配置一个 Google 提供的 JavaScript 库，然后在我们的页面里创建一个 标签作为按钮的容器。

为了加载这个库，我们需要在页面的<head>元素里添加两个标签。首先，需要添加一个<script>元素来引入 Google JavaScript API（gapi）。按照 Google 的建议，这个元素包含 async 和 defer 属性，告诉浏览器在获取这个脚本的时候不要暂停页面渲染，并在页

面渲染完成之后执行这个脚本。

learnjs/4000/public/index.html

```
<script src="https://apis.google.com/js/platform.js" async defer></script>
```

接下来配置这个库，添加一些元数据来识别你所创建的 Google 项目。使用你在身份池配置页里输入的客户端 ID 在页面的<head>里创建一个<meta>元素，像下面这样：

```
<meta name="google-signin-client_id"
  content="ABC123ADDYOURID.apps.googleusercontent.com"/>
```

添加这些之后，定义一个函数，Google API 将用它作为回调函数，在用户登录成功时会调用它。Google 建议把这个函数命名为 googleSignIn。注意，这个函数不能生存在 learnjs 命名空间里。现在创建这个函数并将参数打印到控制台，以便确认它正在被调用。

learnjs/4000/public/app.js

```
function googleSignIn() {
  console.log(arguments);
}
```

现在已经配置好依赖的 JavaScript 库并添加了一个回调函数，接下来就要给 Google 划分一个地方放置它的按钮了。你需要添加一个 到第 3 章创建的 nav-container 元素中。把它放在存放导航条菜单项的<ul>后：

learnjs/4000/public/index.html

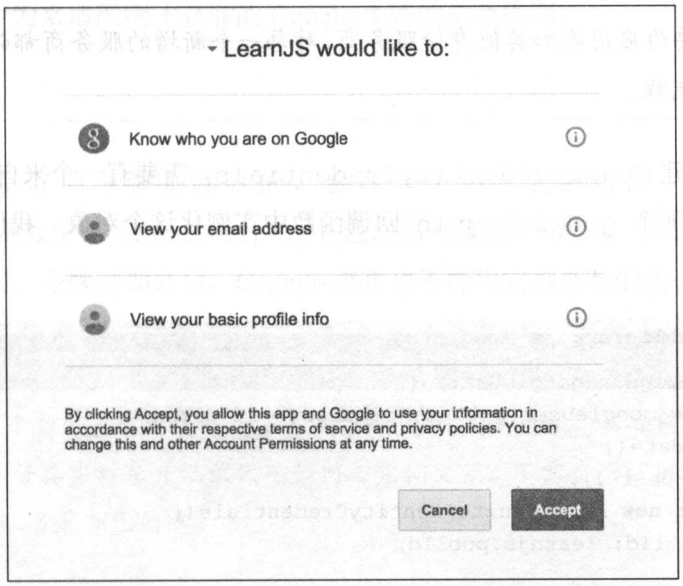
```
<div class='nav-container no-select fixed-top u-full-width'>
  <ul class='inline-list hover-links nav-list six columns'>
    <li><a class='text-lg' href="#">LearnJS</a></li>
    <li><a href="#problem-1">Start</a></li>
  </ul>
  <div class='four columns'>
    <span class='navbar-padding u-pull-right'>
      <span class="g-signin2" data-onsuccess="googleSignIn"></span>
    </span>
  </div>
</div>
```

里的 g-signin2 类告诉 Google 的库使用这个元素作为容器。你还需要把回调函数的名字设为这个元素的 data-onsuccess 属性。因为 Google 的 API 会在这个元素中插入一个 div，所以必须装饰一下它以便能正常显示。这时候可以用 Skeleton 的 CSS 网格把导航条分成两半。创建两个元素：一个 6 列宽，另一个 4 列宽[6]。你还需要添加一个带 padding 和 display 属性的 CSS 规则，像这样：

learnjs/4000/public/index.html

```
.navbar-padding {
  padding-top: 7px;
  display: inline-block;
}
```

当所有这些都就绪，你应该可以加载应用，并且在工具栏看到一个带 Google 标志的登录按钮。单击这个按钮会打开一个新的窗口并通过连接序列重定向，打开之前在 Google 开发者控制台中创建项目时配置的认证页面，如下图所示。



6 http://getskeleton.com/#grid

单击“Accept”按钮会关闭这个窗口。在浏览器的开发者控制台中，你应该可以看到被打印出来的回调函数参数。这意味只我们现在在应用里能获取一个 Google 身份。接下来，我们需要完成这个 googleSignIn 回调函数，用我们接收到的 Google ID 来请求 Cognito 证书，使这些证书对应用生效。

请求 AWS 证书

既然我们已经通过登录程序将应用和 Google 成功地连通，接下来就能用这个身份令牌创建 Cognito 身份了。使用 JavaScript 版 AWS SDK 来创建和配置一个 CognitoIdentityCredentials 对象。这个对象将允许我们的应用获取 AWS 证书，有了这个证书，应用就能作为已认证用户直接访问 Amazon 的 Web 服务。AWS SDK 已经包含在预备工作空间的 vendor.js 文件里了。

如果要为应用添加其他身份服务商，对每一个新增的服务商都必须走一遍同样的流程。

要为用户创建 CognitoIdentityCredentials，需要有一个来自 Google 的身份令牌。在刚才写的那个 googleSignIn 回调函数中实例化这个对象。我们来一步一步地讲解。

learnjs/4103/public/app.js

```
function googleSignIn(googleUser) {
  var id_token = googleUser.getAuthResponse().id_token;
  AWS.config.update({
    region: 'us-east-1',
    credentials: new AWS.CognitoIdentityCredentials({
      IdentityPoolId: learnjs.poolId,
      Logins: {
        'accounts.google.com': id_token
      }
    })
```

```
  })
}
```

你要做的第一件事就是从 `googleUser` 的响应对象中获取 `id_token`。使用这个令牌来创建 `CognitoIdentityCredentials` 对象。你还需要提供身份池的 ID，可以把它像下面这样隐藏在命名空间对象中：

learnjs/4103/public/app.js

```
var learnjs = {
  poolId: 'us-east-1:aa0e6d15-02da-4304-a819-f316506257e0'
};
```

一旦创建了证书，就必须更新 AWS 配置，把它们和身份池所在的 Amazon 可用地域（availability region）包含进来。配置更新之后，需要提交这些修改。然后 Cognito 用户 ID 才会变为可用。Cognito ID（不是 Google ID）是我们用来识别用户的，所有写到数据库的用户记录都要包含它。不过在讨论如何获取 Cognito ID 之前，我们需要为 `googleSignIn` 函数添加更多的行为来适配刚才获取的 Google 令牌的生命周期。

刷新令牌

从 Google 获取的令牌是临时的，会在一个小时之后过期。你也可以通过 `googleSignIn` 函数中 `getAuthResponse` 返回的对象的 `expire_in` 或者 `expire_at` 属性查看过期时间。令牌过期之后，Cognito 就再也不能用它刷新我们的 AWS 证书了。

spy 对象

为了测试[7]这个行为，我用了一个不同类型的 spy：spy 对象。与之前创建的 spy 函数一样，spy 对象代替真实对象并记录代码如何与它们交互。不需要测试用例创建真实的证书，我就能判断证书是否已创建。

通过监听 `CognitoIdentityCredentials` 构造函数，我能返回一个 spy 对象，

7 https://pragprog.com/titles/brapps/source_code

让 appOnReady 函数使用这个 spy 对象。在 Jasmine 中使用 andCallFake 函数，我能立即调用回调函数来模拟所传入的函数。这里把一个异步操作转成同步操作，是我做测试既快又可靠的秘诀。

当令牌过期之后，需要更新令牌。不同的身份服务商刷新令牌的方式不太一样。例如 Google，我们可以使用添加登录按钮时载入的那个 Google API 来刷新令牌。通过调用 gapi.auth2.getAuthInstance().signIn()，就能更新这个令牌，如果用户已经退出，会提示用户重新登录应用。

我们创建一个名为 refresh 的函数来触发更新。可以在 googleSignIn 函数里定义这个函数，因为只有在这个作用域里才会用到它。当 AWS 证书过期时，这个函数（通常在请求响应失败时）还可用于更新身份令牌和刷新 AWS 证书。

Cognito 身份认证的详细内容保存在 localStorage7 里。。

在 refresh 函数里，可以调用 signIn 然后返回这个 Google API 返回的对象。为 login 设置 prompt 选项，这样如果用户已经登录就可以免去不必要的重复认证。带上一个 then 处理函数，在请求成功返回时可以更新 AWS 证书对象。

learnjs/4200/public/app.js

```
function refresh() {
  return gapi.auth2.getAuthInstance().signIn({
      prompt: 'login'
    }).then(function(userUpdate) {
    var creds = AWS.config.credentials;
    var newToken = userUpdate.getAuthResponse().id_token;
    creds.params.Logins['accounts.google.com'] = newToken;
    return learnjs.awsRefresh();
  });
}
```

要留意身份令牌是怎样在不重新创建证书对象的情况下被更新的。最后一步是更新 AWS 证书，我们可以把它封装在一个叫作 learnjs.awsRefresh 的函数里。在下一节

会看到，这个函数返回的对象会被用来将所有的更新请求优雅地串联起来。

基于 Deferred 对象和 Promise 的认证请求

如果修改了 `AWS.config.credentials`[8]对象，需要调用这个对象上 `refresh` 方法来应用修改。这个函数可以传一个回调函数，用来检测刷新是否完成。问题是，这次刷新仅仅是当 Google 令牌过期时需要协调的一连串异步事件中的一个。如果不想编写回调嵌套回调的代码，就要用更好的方式来管理这类请求。

使用 Promise 是解决这个问题的一个方法。Promise/A+标准定义了一种可以用于协调异步事件并将其串联起来的对象。可以异步发送请求，等它们都完成时再关联到一起。我们也可以很方便地把一个请求的结果传给下一个请求。任何符合这个标准的对象都可以与别的实例互相操作。例如，在 `refresh` 函数里调用 `signIn`，返回一个符合标准的对象。现在 Promise 的问题是，并非所有浏览器都提供了它们自己的实现。

在 Promise 被完全支持以前，我们可以使用 jQuery 的 `Deferred` 对象。和 Promise 差不多，`Deferred` 对象是当一个异步操作或者请求完成时执行操作的一种方式。`Deferred` 对象拥有三种状态：pending、resolved 或者 rejected。当我们第一次创建 `Deferred` 对象时，它是 pending 状态。我们可以通过 `done`、`fail` 或者 `always` 等方法来给 `Deferred` 对象传递回调函数。这些回调函数会在 `Deferred` 对象过渡到 resolved(done)或者 rejected(fail)状态时被调用。当 `Deferred` 完成后，可以提供一个值传给 `done` 方法的回调函数。和事件监听器不同，如果 `Deferred` 在传递回调时早就 resolved 或 rejected，它会被立即调用。

Joe 问：

你用 Promise 了吗?

就算知道我有对 Web 标准的偏好，你可能还是会好奇为什么不使用 Promise/A+兼容的对象来管理我们的身份信息。目前 Promise 还没有兼容所有浏览器，降级使用

8 http://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/Credentials.html

Deferred 对象是因为我知道它可行，而且在本书里我会尽量少使用这个工具。

然而，在 jQuery 3 里，Deferred 对象会兼容 Pormise/A+。当它发布之后，即使 ECMAScript 6 还没有兼容所有浏览器，我们也可使用一个 Web 标准来管理串起来的请求。可惜，在写这本书的时候，jQuery 3 还在测试，并不适用于本书。

第一个可以使用 Deferred 的地方就是一个名为 learnjs.awsRefresh 的新函数，它被用来应用 AWS 的配置修改。这个函数可以简单包装我们传给 refresh 函数的回调，当请求成功时将 Deffered 设为 resolved，如果请求失败则设为 rejected。

learnjs/4200/public/app.js

```
learnjs.awsRefresh = function() {
  var deferred = new $.Deferred();
  AWS.config.credentials.refresh(function(err) {
    if (err) {
      deferred.reject(err);
    } else {
      deferred.resolve(AWS.config.credentials.identityId);
    }
  });
  return deferred.promise();
}
```

在上一节，我们通过 googleSignIn 里的 refresh 函数调用了 learnjs.awsRefresh。更新 Google 令牌 ID 之后必须更新 AWS 证书，将所有请求与 Deferred 对象以及 Google API 返回的 Promise 对象串联起来，处理时会更简单。

我们使用 Cognito 提供的 AWS 证书时，总希望这个 refresh 函数好用。在下一节你会看到，如果出现错误时能刷新证书，可以防止我们的用户在令牌失效时采取一些极端手段（如重载应用）。使用返回的 Promise，在更新证书后能方便地重新提交请求。

我们需要在 googleSignIn 函数里做的最后一件事是，初次收到 Google 令牌时应用（apply）我们创建的 AWS 配置。完成之后，我们会构造一个身份对象，它将服务商特有的身份管理细节在我们应用的余下部分中隐藏起来。

创建一个身份 Deferred 对象

为了在应用里管理身份，我们需要用另一个 Deferred 对象。这个对象允许一个视图或者应用的其他部分在用户登录时采取行动，并且可以安全访问个人信息来完成这些操作。我们可以在命名空间里像下面这样为应用创建一个身份 Deferred 对象：

learnjs/4103/public/app.js

```
learnjs.identity = new $.Deferred();
```

googleSignIn 函数的最后一步是调用 learnjs.awsRefresh，并完成这个 Deferred 对象。任何视图或者应用的其他部分如果想在用户登录时做一些事情，可以通过传递一个 done 处理函数给 Deferred 对象来实现。你可以用一个能访问 Cognito ID、刷新函数和其他个人信息（例如用户邮箱地址）的对象来完成 Deferred 对象。

learnjs/4200/public/app.js

```
learnjs.awsRefresh().then(function(id) {
  learnjs.identity.resolve({
    id: id,
    email: googleUser.getBasicProfile().getEmail(),
    refresh: refresh
  });
});
```

使用 Jasmine 做异步测试

在对这段代码做测试时，我使用了 Jasmine[9]的异步支持。给测试函数添加一个可选的 done 参数，就能让测试在调用 refresh 之后等 Promise 链完成再做关于 ID 令牌状态的断言。不过这和使用 jQuery 2 时的问题不一样，这意味着如果升级到 jQuery 3 并开始使用 Promise/A+ Deferred 对象，我的测试就应该通过。

另外，我完全模拟了对 Google gapi 库的调用，因为我不想在测试里引入这个库。由于可以自由使用 Jasmine 的 createSpyObj 方法，所以这种模拟十分简单。如果你对其中的工作原理有疑问，可以到 pragprog.com 查看示例代码和测试代码。

9 http://jasmine.github.io/2.0/introduction.html#section-Asynchronous_Support

这些信息是通过一个身份 Promise 对象获取的，我们不需要依赖任何身份服务商就可以调用它。所以，无论用户是通过 Google、Facebook 还是其他身份服务商登录，应用的其他部分根本察觉不到差别。无论你用什么身份服务商，`Deferred` 完成时的对象结构是肯定是一样的。

既然我们已经为用户创建一个 Cognito 身份，就开始用它做点什么吧。接下来我们会开发个人主页视图（profile view），用户可以通过它来看自己是怎么被接入应用的。这不仅是一个实用的视图，而且让我们得以窥探新的身份管理是如何运转的。

创建个人主页视图

开发现代 Web 应用一个必不可少的部分是，让用户知道系统里记录了他们的什么信息。最基本的方式是把应用中关于用户自己的所有信息展示给他们，这些信息展现在名为个人主页的新视图里。和其他视图一样，我们通过添加路由和视图函数来创建这个视图。首先添加视图函数，像下面这样：

learnjs/4200/public/app.js

```
learnjs.profileView = function() {
  var view = learnjs.template('profile-view');
  learnjs.identity.done(function(identity) {
    view.find('.email').text(identity.email);
  });
  return view;
}
```

接下来，你需要一个视图模板。在这个视图模板里添加一个简单的头和标题。

learnjs/4200/public/index.html

```
<div class='profile-view'>
  <h3>Your Profile</h3>
  <div class='email'></div>
</div>
```

这些都准备就绪之后，可以添加路由。

learnjs/4200/public/app.js

```
var routes = {
  '#problem': learnjs.problemView,
  '#profile': learnjs.profileView,
  '#': learnjs.landingView,
  '': learnjs.landingView
};
```

为了访问我们的新视图，可以添加一个链接到导航条上，和处理题目视图时一样。不要直接从这个视图添加链接，你可以在 appOnReady 中添加一个回调函数到身份 Deferred 对象。这样，用户登录时可以访问个人主页来查看应用使用了他的哪些信息。

learnjs/4300/public/app.js

```
learnjs.appOnReady = function() {
  window.onhashchange = function() {
    learnjs.showView(window.location.hash);
  };
  learnjs.showView(window.location.hash);
  learnjs.identity.done(learnjs.addProfileLink);
}
```

为了在身份 deferred 对象完成时添加链接，首先要创建一个链接模板。

learnjs/4301/public/index.html

```
<div class='profile-link navbar-padding-lg'>
  <a href="#profile"></a>
</div>
```

然后要创建一个 addProfileLink 函数，将 signin-bar 类添加到导航条里包含 Sign-In 按钮的 标签上。在用户登录时，我们就能插入链接模板了。

learnjs/4300/public/app.js

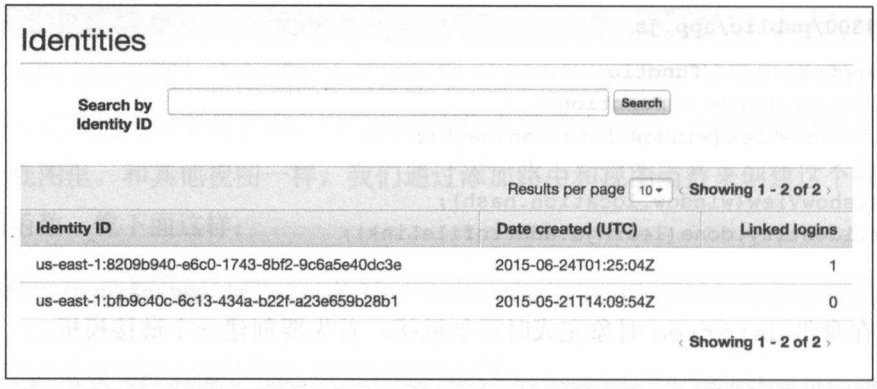
```
learnjs.addProfileLink = function(profile) {
  var link = learnjs.template('profile-link');
  link.find('a').text(profile.email);
  $('.signin-bar').prepend(link);
```

```
}
```

learnjs/4300/public/index.html

```
<div class='four columns signin-bar'>
  <span class='navbar-padding u-pull-right'>
    <span class="g-signin2" data-onsuccess="googleSignIn"></span>
  </span>
</div>
```

如果现在加载应用的话，你应该会看到你的邮箱地址作为一条链接出现在导航条上。点击这个链接就会显示个人主页视图。另外，现在应该有一个已认证的身份加入到了应用的身份池里。如果打开之前在 AWS 控制台里见过的 Cognito 身份查看页面，应该会看到一个关联 1 个登录（a linked login）的新的已认证身份，如下图所示。



成功了！我们的应用现在关联了一个 Google ID，可以用它来获取 AWS 认证证书。我们有一个显示用户详情的视图，能确认我们的身份管理在正常运行。在下一章，我们会用这些证书来发送 Web 服务请求到 AWS。你会学习如何配置 DynamoDB 来安全地存储一个用户专有的数据。

再次部署

既然我们的用户可以登录和查看个人信息，那么可以再部署一次应用了。通常，最好

先验证一下 Google 开发者控制台里项目配置中输入的 URL 是否正确。通过单元测试或者在开发环境里是没办法验证其正确性的。我们可以在测试环境里试试，但是如果 URL 不一样，这么做也没什么用。我们希望在这个 URL 之上构建用户可见的功能之前确认其在生产环境里是可用的。

那么，还是使用在第 1 章“创建一个 S3 存储桶”里用过的`./sspa deploy_bucket`命令来部署应用。完成之后，在浏览器里打开应用，快速浏览一遍。

下一步

现在你已经通过 Amazon Cognito 将 Google 接入到应用，你可能想要研究一些其他主题。为了简单起见，本书没有介绍这些内容，不过你或许想在进入下一章之前自己研究一番。

通过 Facebook 登录

为应用接入 Facebook[10]和接入 Google 差不多。打开 Facebook 开发者门户，然后创建一个新的应用，就会得到一个和 Google 客户端 ID 类似的应用 ID。Facebook 提供一份入门教程，里面介绍了添加登录按钮的大致步骤。

通过 OpenID 登录

OpenID Connect[11]是和 OAuth2 相互竞争的标准，可惜的是它没那么流行。不过，还是有少数服务商提供 OpenID 登录，包括 Salesforce.com 和 Google。Cognito 有接入兼容 OpenID 的供应商的机制，所以对我们的应用而言这也是一个可选方案。

扩展你的测试环境

正如我们的 S3 存储桶一样，在创建测试环境的时候，你可能希望为它添加一个身

10 http://docs.aws.amazon.com/cognito/latest/developerguide/facebook.html

11 http://docs.aws.amazon.com/cognito/latest/developerguide/open-id.html

份池。这个过程的步骤和本章的一样，只不过要为你的身份池设置不同的名字，但是不需要创建另一个 Google 应用。

开发者认证身份

不需要让 Amazon 保存你的身份信息就可以通过 Cognito 获取临时的 AWS 证书。有了开发者认证身份[12]，就可以开发自己的认证程序，而且还能用 Cognito 来颁发访问 AWS 资源的证书。如果你要整合现有的认证系统，也许会想参考这个方案。

在下一章中，我们会优化应用，保存用户提供的答案。我们会连接 DynamoDB，为本章创建的 Cognito 角色添加一个 IAM 策略，你就会明白为什么数据验证比你想象的更复杂。

12 http://docs.aws.amazon.com/cognito/devguide/identity/developer-authenticated-identities/

第 5 章
使用DynamoDB存储数据

到目前为止，我们搭建的所有东西都与将要做的事情相关。我们已经创建了一个完全只有客户端，没有中间层的应用。我们把这个应用连接到 Amazon Cognito 上，创建了一个可以用来直接访问 Amazon 其他服务的身份。现在，是时候连接到这些服务了。

不管是单页还是多页应用，存储数据是大部分 Web 应用都要做的事情。在本章中，你将看到无服应用是如何与 Amazon 的 DynamoDB 数据库服务进行交互的。随着我们对这种方式的进一步探究，你会开始理解其中的权衡关系。你将了解哪些是保证数据完整性的必要步骤，学习如何构造数据以达到一致、可扩展的查询性能。在本章最后，你会知道如何以及什么时候从浏览器中直接和数据库进行交互。

在开始读取和写入数据之前，需要理解与自己在打交道的是什么服务。DynamoDB 是一个强大的工具，但了解了它的长处和短板才能有效地使用它。正如下一节我们将看到的一样，一切始于数据结构。

使用 DynamoDB

DynamoDB 的根本在于其键值对存储。在 2014 年的假期购物季中 Amazon 发生了一系列严重的服务中断事故[1]，之后它就创建了 Dynamo。现在，这个系统的下一个版本——DynamoDB，作为一个 Web 服务发布，你可以将它用于自己的项目中。

对于 Amazon 来说，DynamoDB 的目标之一就是建造一个能够无限扩展的数据库，而且似乎它已经实现了[2]。对于一个 DynamoDB 表而言，其中的数据量以及这个表大小，并没有实际的限制。换句话说，只要你一直付钱，Amazon 就会一直为你扩容。

除了令人惊叹的容量之外，DynamoDB 还有其他一些特点。为了理解 DynamoDB 是如何组织你的数据的，你必须深入底层，理解它的一些工作原理。例如，为了获得 Amazon 承诺的可预测的查询性能，你就要理解 DynamoDB 如何处理主键和哈希。如果你之前只用过关系型数据库，这里有些地方可能会有点反直觉。

理解 DynamoDB 的键和哈希

和关系型数据库一样，DynamoDB 用表来组织数据。但表的内容却有点不同。DynamoDB 让你存储带有任意个属性的对象（item），而不是按行存储指定列数的记录。每个属性有一个名字和一个值。一个 item 唯一必需的属性是用于识别它的主键属性。主键可以是一维也可以是二维的。

多维键

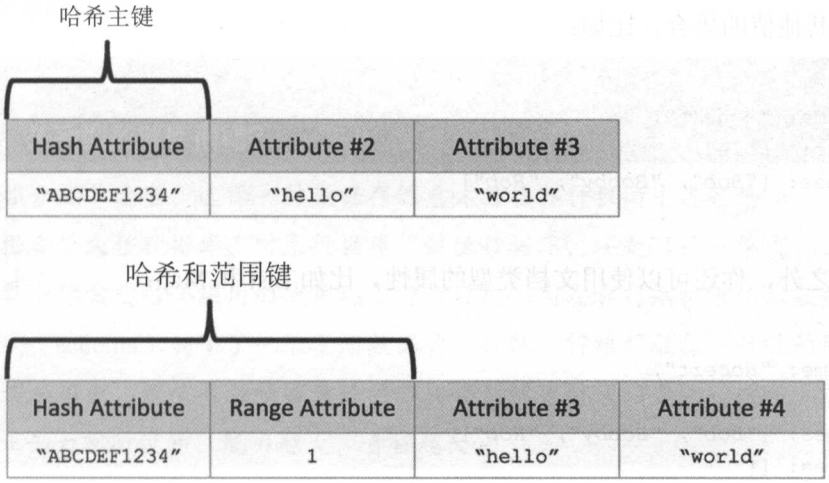
DynamoDB 将多维键映射到一个值上。多维键就是由多部分构成的一个键。例如，一个三维键可能包含一个属性名、一个 SHA-1 字符串和一个整型时间戳。所有的这些部分合在一起唯一映射成一个值。

1 http://www.allthingsdistributed.com/2012/01/amazon-dynamodb.html

2 http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Limits.html

如果主键只有一个维度属性，那么它的值就必须是唯一的。Amazon 称这种一维键为一个哈希主键（hash primary key）。哈希主键以无序的形式存储，它的值可以是字符串、数字或者字节流（binary）。

如果主键有两个维度属性，则第一个称为哈希属性（hash attribute），第二个称为范围属性（range attribute），可参见下图。在这种情况下，表中的 item 按照范围属性的顺序进行存储。如果你在关系型数据库中用过组合键，这种二维键应该听起来比较熟悉。由于 DynamoDB 采用这种结构化方式，用范围属性进行查询是非常高效和可扩展的。当查询这个属性时，你可以使用条件来确保范围属性值大于或者小于给定值，或者在几个值之间。



DynamoDB 用作文档数据库

DynamoDB 本质上是个键值数据库，但它可以用作文档数据库，这也是我们将要在应用中使用的方式。只要我们数据模型中的对象符合 DynamoDB 强制执行的限制条件，就可以用它来存储 JavaScript 对象，而无须将它们映射到数据库行或者从行中解析。

DynamoDB 的 item 最大可以到 400 KB。这个大小包括了属性的名和值，但不包括在 JSON 格式中显示的任何括号和逗号。我们应用的文档对象应该不到 400 KB。

除了 item 的大小有限制以外，属性名不能超过 255 字节。要记住，Unicode 字符串在 UTF-8 中是被编码成多个字节的，所以如果使用 UTF-8，字符的长度和字节数不一定相同。尽管我们会想要避免在属性名称中使用模糊的 Unicode 字符，但仍希望确保其长度是小于 255 个字节的。

DynamoDB 的属性值支持多种类型。属性可以是标量类型，比如 Number、String、Binary 或者 Boolean。在请求中，Binary 类型的值应该使用 Base64 编码。属性值可以是 NULL，但不能是空的（零字节）。

DynamoDB 也支持多值类型，比如 String Set、Number Set 和 Binary Set。所以属性的值可以是其他值的集合，比如：

```
{
  firstName: "Robert",
  lastName: "Dole",
  nickNames: ["Bob", "Bobby", "Rob"]
}
```

除此之外，你还可以使用文档类型的属性，比如 List 和 Map：

```
{
  firstName: "Robert",
  lastName: "Dole",
  nickNames: ["Bob", "Bobby", "Rob"],
  addresses: [{
    label: "home",
    street: "123 Fake St.",
    city: "Anytown",
    state: "NY",
    postalCode: "12345"
  }, {
    label: "work",
    street: "456 Phony Ln.",
    city: "Nowheresburg",
    state: "CA",
    postalCode: "54321"
  }]
}
```

DynamoDB 的表不需要有模式（schema），在一个表中每个 item 的属性可以是不同的。不像关系型数据库，这意味着你的文档格式更改不需要切换表结构。根据改动的类型，你可能根本不需要对现有数据进行任何修改。例如，添加一个属性通常情况下完全可以在应用中实现。需要的话，你还可以在数据库中同时保存不同格式的数据，然后依靠你的应用来区分它们。

以这样的方式使用 DynamoDB，我们就可以将数据模型中的 JavaScript 对象直接写入 DynamoDB 中。如果我们需要查询出这些记录，得到的结果将会是可以直接插到我们数据模型中的 JavaScript 对象。通过调整数据模型来适应服务的特点，我们可以轻松利用 DynamoDB 带来的好处。

Joe 问：

所以，关系型数据库很糟？

就像大部分技术选型，选择一种数据存储意味着在各种权衡中进行选择。不管是选择关系型数据库、文档数据库、对象数据库、键值数据库，还是只是一堆文件，每种方案都各有利弊。你需要结合应用的使用场景进行考虑。用表和行结构来组织数据可以便于分析。模式（schema）确保了记录之间数据的一致性。将维护数据一致性的职责从数据库中迁出到应用中，意味着必须留意你的数据模型随着时间的变化。这种方法可能非常强大，但正如大家所说的，能力越大，责任越大。

强一致性和最终一致性

如果把数据写入 DynamoDB 后，立即尝试读它，你可能不会得到和写入时相同的值。这个行为称为最终一致性（eventual consistency），在很多分布式数据库中这是非常常见的。理解为什么会发生这种情况以及如何避免是非常重要的，尽管在有些场景下，你可能不想这么做。

DynamoDB 只有一种基本的写入数据的方式，但有两种读取数据的方式。你可以执行最终一致性读取或者是强一致性读取。到底选择哪一种，取决于使用场景，两种方式各自

都有取舍。

强一致性读取就是你通常认为的从数据库中读取数据的方式。这个操作返回“能反映之前 DynamoDB 成功返回的写操作更新的最新数据”[3]。然而，这种读取方式很可能会因为网络中断导致失败，而且还有一点可能更重要，它花的钱更多。正如我们后面会介绍的，你需要购买更多的容量来实现强一致性读取。

这意味着一般我们会把应用设计为能处理最终一致性读取。你可能会问的第一个问题是，读取的是多大程度上的最终数据？DynamoDB 的文档写到，“数据副本间的一致性同步一般可以在一秒之内完成”[4]。在实践中，这意味着对于需要人工交互的操作是可以依赖最终一致性读取的数据的。任何用编程完成的工作都需要用到强一致性读取，但我们可以在应用逻辑中绕开这个问题，因为在我们的应用中写入数据库的任何内容都会首先存在于数据模型中。

既然你理解了 DynamoDB 的能力及其限制，下面就该创建一张表进一步学习了。你将学习如何定义我们的主键，分配读/写容量和创建索引来提升查询性能。你将会了解我们如何通过定义安全策略，把 Cognito 权限认证集成到表中，来实现只允许获得授权的用户进行访问。

创建表

要创建这张表，我们可以使用预备工作空间中的 `sspa` 脚本。正如第 4 章中我们创建的身份池一样，脚本中的一个 action 将会创建和配置所有必需的资源。脚本用的两条 AWS CLI 命令是 `dynamodb create-table` 和 `iam put-role-policy`。另外，和之前一样，我们将从头到尾看一遍脚本，看看它可以为我们做什么，这样它就不再显得那么神秘了。

3 http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.DataConsistency.html

4 https://aws.amazon.com/dynamodb/faqs/#Read_Consistency

在预备的工作空间中有一张预配置好的表，它有一个配置，你可以直接拿来用或者做一些修改。这个配置存放在 conf/dynamodb/tables/learnjs/目录下。打开里面的 config.json 文件看看：

learnjs/5000/conf/dynamodb/tables/learnjs/config.json

```
{
  "AttributeDefinitions": [
    {
      "AttributeName": "problemId",
      "AttributeType": "N"
    },
    {
      "AttributeName": "userId",
      "AttributeType": "S"
    }
  ],
  "KeySchema": [
    {
      "KeyType": "HASH",
      "AttributeName": "userId"
    },
    {
      "KeyType": "RANGE",
      "AttributeName": "problemId"
    }
  ],
  "ProvisionedThroughput": {
    "ReadCapacityUnits": 5,
    "WriteCapacityUnits": 5
  }
}
```

这个文件被传给 dynamodb create-table 命令，它指定了表的配置。这个配置里的三个顶级属性——AttributeDefinitions、KeySchema 和 ProvisionedThroughput，都是必需的参数。接下来，我们讲解这些设置的含义，然后看看那些我们还没指定的可选设置。

属性和键值

DynamoDB 中的属性不仅有名字和值，它们还有类型。正如你之前看到的，它们可以是像 String 和 Number 这样的简单类型，也可以是像 Map 和 List 这样的复杂类型。当写数据到表中时，JavaScript 的 AWS SDK 试图基于被保存的对象中的数据，为属性检测到一个合适的类型。

然而，如果你知道它们会用作什么，可能希望在使用之前就定义它们的属性类型。在我们的例子中，我们知道需要一个数值型 `problemId` 属性，以及一个字符串型 `userId` 属性。这些类型通过 `AttributeType` 属性和一个代表类型的字符串值来定义。在 AWS 文档[5]中可以查到可用的类型和对它们的描述。在这里，我们用“`N`”表示数值类型，用“`S`”代表字符串类型。

但是注意，你不需要提前确定属性的定义。DynamoDB 不需要固定的记录模式（record schema），你可以为写入到数据库的任意记录添加新属性。你还要将属性放入 `config.json` 文件中，以便创建表，但是不需要为记录添加所有你希望有的属性。AWS SDK 会在你写入带有新属性数据记录的时候，自动检测属性类型。正如上一节所述，DynamoDB 为表的主键的结构提供两种选择。在本例，我们希望添加一个排序键（也称为范围键）。通过使用 Cognito 身份 ID 作为哈希键，使用问题编号作为范围键，我们将不仅能提供对这个表中 item 的快速检索，而且可以限制用户只能访问他们自己创建的数据。

`Config.json` 中的 `KeySchema` 属性定义了我们想要的表的主键设置。添加两个对象到这个数组中：一个给 `HASH` 键，一个给 `RANGE` 键，我们就可以指定将由数据里的哪些属性组成我们的多维键。注意，键的顺序很重要，第一个必须是 `HASH` 键，`RANGE` 键（如果有的话）必须是第二个。尽管这个属性是个数组，但是不允许有多个 `HASH` 键和 `RANGE` 键。

---

5 http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_AttributeValue.html

预设吞吐量

对于所有 Web 服务，包括 DynamoDB 在内，有一点很关键，即要了解它们是如何收费的。有些费用是和数据存储和数据传输相关联的，在创建表的时候不需要关心。现在需要搞清楚的是，在我们的表里面应该如何设置 `ProvisionedThroughput`。

对于 DynamoDB，要提前购买容量（capability）来执行读和写操作。这个额度是按照读和写操作单位容量来计算的，你可以根据应用的需要增加或者减少容量。每个读单位容量提供每秒执行一次强一致读操作的能力，每条 item 不能超过 4 KB。如果使用最终一致性读，一个读单位容量每秒能执行两次这样的读操作。一个写单位容量允许你每秒写一个 item，其大小不能超过 1 KB。这些单位容量是线性计算的，比如一次强一致性读 40 KB 的 item，要用 10 个读单位容量，在 1 秒内写入 5 个 1 KB 的 item 将会消费 5 个写单位容量。

你可以随时修改预设吞吐量来满足应用的需要。对于每个表，通过 AWS 控制台或者 API 最多可以预设 40,000 个读单位容量。超过这个数值，就需要联系 Amazon 客服。Amazon 声称 DynamoDB 可以无限扩展，但是你必须提前通知他们。

在预设吞吐量的时候，需要考虑应用做查询的随机程度如何。Amazon 建议选择那些最终将你的查询负载分布到整个哈希空间上的主键。这是因为 DynamoDB 会把总的键空间切分为不同分区（partition），然后为每个分区[6]分配一部分你的数据（和吞吐量上限）。所以，你可能很自然地认为你的预设吞吐量会应用到整张表。然而，如果应用对表中键的查询不是均等的，尽管没有超过分配的总容量，但是你可能仍然会遇到容量问题。

如果应用在运行时超出了分配的容量，它发送给 DynamoDB 的任何请求都会返回 `ProvisionedThroughputExceededException`。我们的主键是 userId，所以对于应用来说，这就意味着如果我们有一个用户执行的读写操作比例不均衡，他可能会碰到容量错误而其他用户那里却没有什么问题。到底把这一点当作是一个 bug 还是一种功能，就要看你怎么理解了。

---

6 http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GuidelinesForTables.html#Guidelines-ForTables.Partitions

为了给我们的新表定义预设吞吐量，需要在 config 对象中设置 `ReadCapacityUnits` 和 `WriteCapacityUnits` 属性。预备工作空间中的 config 例子把这两个属性值设置为 5，但 AWS Free Tier 为你的所有 DynamoDB 表提供了最多 25 个免费读/写容量单位。如果你为生产和测试环境创建了两张表，可能希望在两个表中分配这些额度，否则就可以把这个值设置成 25，而不会产生任何费用。

你可以将 Free Tier 提供的吞吐量分割成不同份额，分配给多张表。

既然你理解了什么是吞吐量上限（throughout capacity），那么接下来就可以把你对这个值的改动保存到 `config.json` 文件中。由于这个值可以随时修改，所以第一次设置得正确与否并不是那么重要。然而，如果你在应用中发现错误，肯定希望立即修改这些值。你可以使用 AWS CLI 工具修改，或者在 AWS Web 控制台中调整表的设置。

这个表有几个设置我们在 `config.json` 中尚未定义。在继续往下讨论之前，我们先来看看创建 DynamoDB 表格时你会看到的另一个选项。尽管你现在可能用不上它，但当你的应用扩容时，它可能会变成应用不可缺失的一部分。

二级索引与查询 vs 扫描

要从一张 DynamoDB 表中取出数据，主要有两种方式。一个查询能有效地根据主键值来挑出表的 item——通过 hash，或者 hash 和 range，具体情况取决于这个键的结构。当使用 range 键时，查询可以很快地选择表中符合查询条件的一部分 item。

另一种从表中取数据的方式是使用扫描（scan）。扫描操作会评估表中的每一条 item，它可能比查询操作要慢很多。默认情况下，扫描操作返回表中的所有 item（不超过 1 MB 的数据），但你可以提供筛选器表达式来缩窄结果。然而，就像正则扫描操作一样，这个表达式将对表里的每条 item 进行评估。

如果表包含的数据太多，对一张表执行扫描操作可能会很慢，即使最后得到的结果只

包含几条 item。为了解决这个问题，你可以使用二级索引[7]的查询。有两种类型的二级索引：全局的和局部的。

一个全局二级索引包含了表中所有数据的副本，但它是通过不同主键来索引的，因而能快速访问。局部二级索引使用表相同的 hash 键，但它基于表中 item 的一个属性提供了一个额外的 range 键。全局二级索引可以有一个 hash 或者一个 hash 和 range 主键。一个全局二级索引的主键可以是表中的任何一个属性。

全局二级索引需要有自己的预设吞吐量。

在创建表时只能创建局部二级索引，但是如果有需要你可以稍后创建全局二级索引。当你的应用规模增长时，要记住这一点。现在暂时不需要为应用创建任何二级索引，但随着数据模型演进和表的规模开始扩张，我们可能需要添加二级索引来实现可承受的性能指标。创建一个或者多个全局二级索引，能快速而简单地解决查询方面的性能问题。

既然你已经理解表的配置中的所有选项，是时候创建这张表了。你将需要指定配置目录，以及包含允许访问表的用户的身份池名称，如下所示：

```
learnjs $ ./sspa create_table conf/dynamodb/tables/learnjs/ learnjs
```

需要提供身份池名称，是因为 `sspa` 脚本还会创建必需的 IAM 策略来允许这些用户访问。在下一节，我们将会讲述生成的策略以及 Cognito 和 DynamoDB 是如何协同工作来保障对数据库直接访问时的安全的。

授权访问 DynamoDB

传统 Web 应用经常在业务逻辑中强制使用数据访问策略。这往往意味着当执行一个数据库查询来获取用户描述信息时，基于这个信息和请求类型与内容，要么拒绝请求，要么

7 http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/SecondaryIndexes.html

执行它。这一类的检查必须在应用服务器这样的安全环境中执行，才是有效的。

正如在第 6 章将看到的，对于无服应用而言，是可以在业务逻辑中执行任意安全检测的，但还有一个方案是用一种完全由数据驱动的方式来控制访问。例如在我们的应用中，可以使用 DynamoDB 提供的细粒度访问控制（fine-grained access control）设施[8]。

在开始推行访问规则之前，先搞清楚我们希望它们做什么。我们希望用户能够在 `learnjs` 表里创建文档。这个表将包含他们往应用里输入的所有正确答案。我们还希望他们能够读取创建的任何文档，而且只能更新他们自己拥有的文档。用户应当无法访问其他用户的文档（至少不能直接访问），他们应该无法访问 DynamoDB 的任意其他表。

为了推行这些规则，`sspa` 脚本创建了一个 IAM 策略。这个脚本添加了一个内联策略到授权用户角色上，该角色是与你创建表时指定的身份池相关联的。以下是一个 IAM 策略的例子：

learnjs/5001/conf/dynamodb/tables/learnjs/role_policy.json

```
{
  "Version": "2012-10-17",
  "Statement": [{
      "Effect": "Allow",
      "Action": [
        "dynamodb:BatchGetItem",
        "dynamodb:BatchWriteItem",
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:UpdateItem"
      ],
      "Resource": ["arn:aws:dynamodb:us-east-1:730171000947:table/learnjs"],
      "Condition": {
        "ForAllValues:StringEquals": {
          "dynamodb:LeadingKeys": ["${cognito-identity.amazonaws.com:sub}"]}
    }
```

8 http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/FGAC_DDB.html

```
    }]
}
```

在这个策略的 Action 节里，可以看到我们给所有关联的 DynamoDB 操作都授予了权限。Resource 条目限制了这些权限只能应用于我们的 DynamoDB 表，其表名由 ARN 识别。这个策略的 Condition 条目使用了一个 substitution 变量从请求中获取 Cognito ID。这确保了只有授权的 Cognito 用户才可以访问这张表，而且他们只能访问自己创建的文档。

表和策略都创建好之后，授权的 Cognito 用户就拥有了向 DynamoDB 表读/写记录的权限。只有该用户的记录才是可访问的。现在我们需要做的就是将这个特性添加到应用中。

保存文档

既然已经创建了表，并授权应用与它进行连接，就让我们保存来一些数据看看吧。当用户输入一个正确答案时，我们就保存起来。如果他们之后又回到这一题，我们就把他们之前输入的这个正确答案显示出来。为了将这个功能添加到应用中，首先需要改变问题视图的点击事件处理函数来抓取用户输入，并创建一个对象。然后我们将看到这个对象是如何保存到 DynamoDB 中的。

当写入对象到 DynamoDB 时，可以使用 DynamoDB.DocumentClient 来和 DynamoDB 像文档数据库一样交互。写入的这些对象被表示成 item，并且以该对象的特征（property）作为其属性（attribute），属性值可以是像 String 和 Number 这样的原始值，也可以是像 Object 和 Array 这样的复杂值。每个 item 的属性都可以不同。

DynamoDB.DocumentClient 上的方法有一个优点，就是它们会返回一个 AWS.Request 对象。这个对象封装了一个异步请求。你可以在对象上附一个事件监听器，当请求成功或者失败时（或者一些其他事件发生时）通知该对象。

接下来，看看如何利用这个优点。我们将创建一个通用函数来发送封装了错误处理的数据库请求。然后使用这个函数保存一个 item 到 DynamoDB 上。当然，你可以通过许多不

同的方式和 DynamoDB API 交互，但是这种方式将帮助你更好地理解 DynamoDB 能做什么，了解自己日后需要能处理哪些常见的错误。

一个 fail-safe 的数据访问函数

这个函数名叫 sendDBRequest，提供对 DynamoDB 的访问。这里有很多东西可讲，所以我们从头到尾看一遍：

learnjs/5200/public/app.js

```
line 1 learnjs.sendDbRequest = function(req, retry) {
   -   var promise = new $.Deferred();
   -   req.on('error', function(error) {
   -     if (error.code === "CredentialsError") {
   5       learnjs.identity.then(function(identity) {
   -         return identity.refresh().then(function() {
   -           return retry();
   -         }, function() {
   -           promise.reject(resp);
  10         });
   -       });
   -     } else {
   -       promise.reject(error);
   -     }
  15   });
   -   req.on('success', function(resp) {
   -     promise.resolve(resp.data);
   -   });
   -   req.send();
  20   return promise;
   - }
```

首先，创建一个新的 Deferred 对象。在 sendDbRequest 函数中创建并返回这个对象，就能将它和其他异步请求绑定。一旦创建了 Deferred，就可以通过 on 方法附上两个事件处理函数到请求中——一个用于 error，另一个用于 success。

一个成功的响应是很容易处理的。正如以上代码的第 17 行，我们用响应中的数据解决

（resolve） Deferred 对象。处理错误事件则有点复杂。如第 4 章所述，AWS 的认证证书（credential）在 1 小时之后就会过期，你必须刷新它们。如果对 DynamoDB 的请求失败，返回 CredentialsError[9]，我们就知道证书需要刷新。在第 4 章中，我们为 identity 对象添加了一个函数来刷新证书。检查错误返回对象的这个错误码，我们才知道该不该在此处使用它，就像代码第 4 行做的那样。

在错误回调函数中调用 identity.refresh 能刷新身份服务商的令牌以及 AWS 证书。因为 identity.refresh 返回一个 promise，一旦证书被刷新，我们就可以通过传递给 then 方法的一个回调函数重新提交请求，调用第 7 行的 retry 回调函数。

如果 DynamoDB 请求因为任何其他原因失败，在第 13 行中直接拒绝 Deferred 对象即可。拒绝或者解决这个带 AWS 响应的 promise，使调用者在必要时能采取对应的行动。一旦两个事件处理器函数都注册了，最后一步就是真的发送请求，你可以使用 send 方法来完成。SendDbRequest 的调用方可以使用返回的 promise 来表示该请求是成功还是失败。

创建和保存一个 item

既然有了这个通用函数，我们就可以构建另一个函数来创建 DynamoDB item 并且保存了。调用函数 saveAnswer，如下所示：

learnjs/5200/public/app.js

```
learnjs.saveAnswer = function(problemId, answer) {
  return learnjs.identity.then(function(identity) {
    var db = new AWS.DynamoDB.DocumentClient();
    var item = {
      TableName: 'learnjs',
      Item: {
        userId: identity.id,
        problemId: problemId,
        answer: answer
```

9 http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ErrorHandling.html#APIError

```
    }
  };
  return learnjs.sendDbRequest(db.put(item), function() {
    return learnjs.saveAnswer(problemId, answer);
  })
 });
};
```

saveAnswer 函数创建了 item 对象，添加了必需的字段，比如 Cognito ID。所以，这里要做的第一件事情就是等待认证 Deferred 函数完成。我们需要将这条 item 的 userId 设置为当前用户的 Cognito ID。如果这个值和用户在 Cognito 注册的 ID 不一致，请求将会失败。

使用请求回调函数

除了为 DocumentClient.put 返回的 Request 对象添加事件监听器，你也可以直接传递一个回调函数作为第二个参数。这会使创建 sendDbRequest 函数变得更难，所以我选择了使用请求对象。但是，你可以通过许多不同的方式使用这个 API，本章的例子意在激励大家使用，以及解释它的用法，并不代表最佳实践。

一旦 item 被创建，就可以调用 DocumentClient.put 来创建一个 Request 并传给 sendDbRequest。记住，sendDbRequest 需要一个回调函数在证书错误的时候重试请求。我们的重试逻辑目前来说非常简单：用当前的参数重新调用当前的函数来重试。

重构：将 answer 的声明提取出来，放入周边的函数（surrounding function）。

最后一步是在 checkAnswerClick 处理函数内使用这个新函数。你要取出 answer 的变量声明，放入到周边的函数中，以便它能够被访问。然后需要将题目编号与 answer 的输入值一起，传给 saveAnswer，如下所示：

learnjs/5200/public/app.js

```
function checkAnswerClick() {
  if (checkAnswer()) {
    var flashContent = learnjs.buildCorrectFlash(problemNumber);
```

```
    learnjs.flashElement(resultFlash, flashContent);
    learnjs.saveAnswer(problemNumber, answer.val());
  } else {
    learnjs.flashElement(resultFlash, 'Incorrect!');
  }
  return false;
}
```

将 saveAnswer 和 sendDbRequest 函数的行为隔离，不仅让我们有机会重用 sendDbRequest，而且让代码更加易读。它还创建了一个测试边界，让测试更简单。既然保存了数据，那么接下来就开始读取它。

读取文档

我们放了一个文档到 DynamoDB 中，下面把它重新读取出来。我们可以重用 sendDbRequest 来创建一个函数，异步获取一个保存过的答案。代码应该是下面这个样子：

learnjs/5400/public/app.js

```
learnjs.fetchAnswer = function(problemId) {
  return learnjs.identity.then(function(identity) {
    var db = new AWS.DynamoDB.DocumentClient();
    var item = {
      TableName: 'learnjs',
      Key: {
        userId: identity.id,
        problemId: problemId
      }
    };
    return learnjs.sendDbRequest(db.get(item), function() {
      return learnjs.fetchAnswer(problemId);
    })
  });
};
```

正如 saveAnswer 一样，我们需要从获取 Cognito ID 开始。一旦有了它，就可以创建请求对象，此对象和我们刚才保存的 item 有相似的结构。其主要区别在于，我们没有指定要保存的 item，而是创建了一个对象来代表要获取的 item 的键。

一旦我们创建了这个对象，就可以调用 get 来创建一个 AWS Request 对象，并将其传给 sendDbRequest。和前面一样，我们需要提供一个重试函数，以防证书过期。

有了这个重试函数，现在就可以尝试取出对于当前题目之前保存的答案了。在 problemView 函数里，需要调用 fetchAnswer，并且为得到的 Deferred 附上一个处理函数。当 Deferred 对象 resolve 的时候，我们就得到题目的答案。可以把答案放在视图中的 answer 文本区内。

learnjs/5300/public/app.js

```
learnjs.fetchAnswer(problemNumber).then(function(data) {
  if (data.Item) {
    answer.val(data.Item.answer);
  }
});
```

现在，当用户得到一道题的正确答案时，答案会被保存下来。如果该用户之后再遇到这道题，或者重新加载了应用，他们的正确答案会被恢复。对于不想登录系统来保存答案的用户，也可以像以前一样使用应用。如果在任何时刻某个登录用户的 AWS 证书过期，我们将会刷新它们，并继续。

到这里，我们有了一个很大的进展，不过在部署这个改动之前，我们还需要讲一件事情。尽管对本例来说无关紧要，但是这一点很重要，即了解对写入到 DynamoDB 的数据进行验证时有哪些方法。你将会看到，在无服应用中传统方法并不管用，所以我们才会另辟蹊径来解决这个问题。

数据访问和验证

在传统 Web 应用中，应用服务器一般执行对用户请求的验证。比如，在往数据库写入一条记录之前，应用服务器可能先检查这个记录，看看它是否存在与其他关切（concern）相关联的问题，从用户体验到安全和数据完整性。

在无服 Web 应用中，你需要把这些关切分开。出于安全原因而做的验证操作不能在 Web 客户端执行。任何我们应用能做的，用户也能做，因为浏览器环境是完全受控于用户的。例如，在大多数 Web 浏览器的开发控制台中都能轻易修改应用的代码，恶意用户还能通过这种方式往 DynamoDB 写数据。即使不在浏览器中实现这些功能，如果有人能获得从 Cognito 得到的证书并通过 HTTP 发送请求，唯一能阻止他们向我们的表写数据的保障就只有我们的安全策略了。

为了得到良好的用户体验，Web 客户端可以对数据进行验证，比如确保用户正确地输入一个手机号或者邮政编码。在本例中，应用在保存答案之前会验证其正确性。但无论何种情况，我们都是为了用户的利益而不是系统的利益做这种验证的。除非是可以由 Web 服务强制执行的约束，否则对于能够从客户端直接写入的数据无须做任何假设。

尽管可以在应用中执行用户体验的验证，我们必须在 Web 服务中执行所有的数据访问检查。DynamoDB 为验证这样的请求提供了一些工具。展开我们 IAM 策略的 `conditions` 子句，就可以检查请求的属性。

定义“有效的”数据

有些验证是为了确保好的用户体验，而有些是为了安全。把二者混为一团是非常危险的。你很容易陷入一种幻觉——认为自己数据是安全的，因为你对它们做过验证，而实际上这些验证只能保证数据一致性或者好的用户体验。保证了用户名小于 255 个字符，并不能保证这个用户名不会成为 SQL 注入攻击。把这些重要问题分开来考虑，才能保障它们都能被有效地解决。

举个例子，在我们的应用中，可以让获得授权的用户统计其他用户的答案数。然后，应用执行一个带筛选条件的 scan 请求，来确定某道题有多少用户已经做了回答，或者查清楚哪道题回答的人最多。为了实现这个功能，我们需要创建一个新的策略来授权这次访问，使用 IAM Web 控制台或者 AWS 命令行界面把它添加到已授权的 Cognito 用户角色中。这个策略如下所示：

learnjs/5500/policy/table_policy_condition.json

```
{
  "Version": "2012-10-17",
  "Statement": [{
      "Effect": "Allow",
      "Action": ["dynamodb:Scan"],
      "Resource": ["arn:aws:dynamodb:us-east-1:730171000947:table/learnjs"],
      "Condition": {
        "ForAllValues:StringEquals": {
          "dynamodb:Select": ["COUNT"]
        }
      }
  }]
}
```

ForAllValues::StringEquals 意味着 DynamoDB 确保每个键，比如 dynamodb:Select，匹配请求的指定值。因为我们使用的是 ForAllValues 而不是 ForAnyValue，所以这些检查是使用逻辑“与”连接的，我们这里添加的任何键都需要有值与之匹配。我们可以在这些子句里使用一堆不同的策略键。有些是 DynamoDB 请求[10]特有的，而有些则可以用于任何请求[11]。例如，通过 dynamodb:Attributes 键，我们可以限制用户访问表里的某些属性，或者通过 aws:SourceIp 键限制对指定范围 IP 地址的访问。

将这个策略添加到 IAM 的授权 Cognito 用户角色上，我们就能从应用中发送一个 scan

10 http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/UsingIAMWithDDB.html#IAMPolicyKeys

11 http://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_elements.html#Condition

请求来统计符合给定筛选条件的答案文档。在接下来的代码中可以看到，我们使用 sendDbRequest 函数来发送这个请求，其他的 DB 访问函数也使用这个 sendDbRequest 函数。注意，使用了 FilterExpression 和 ExpressionAttributeValues 项，就限制了某个特定问题 ID 的结果范围。

learnjs/5500/public/app.js

```
learnjs.countAnswers = function(problemId) {
  return learnjs.identity.then(function(identity) {
    var db = new AWS.DynamoDB.DocumentClient();
    var params = {
      TableName: 'learnjs',
      Select: 'COUNT',
      FilterExpression: 'problemId = :problemId',
      ExpressionAttributeValues: {':problemId': problemId}
    };
    return learnjs.sendDbRequest(db.scan(params), function() {
      return learnjs.countAnswers(problemId);
    })
  });
}
```

然后这个请求生成了一个带 Count 属性的响应对象，指定了答案文档的数量。这些文档有一个 problemId 属性，与我们传给 countAnswers 函数那个值相等。没有制订这个策略，发送这个请求就会导致一个错误，如下所示：

```
User: «Role ARN» is not authorized to perform:
  dynamodb:Scan on resource: «Table ARN»
```

虽然这个工具很有用，但是有些安全问题不能使用 Condition 子句来解决。例如，现在还不能使用一个策略来验证数据本身。我们只能实施一些规则，限制请求的类型、它从哪里来、谁发送的，等等。尽管用这种方式来控制访问确实非常简单，但是在某些情况下仍然需要用到比 IAM 策略更加复杂的验证方式。

在那些情况下，你可以创建一个自定义 Web 服务来处理请求。这个 Web 服务可以在写入数据之前先进行检查，或者针对请求结果执行任意筛选。用 Amazon 的 Lambda 服务实

现这些是很简单的，这也是下一章将要讲的内容。不过在此之前，我们应该部署创建的新功能。

重新部署

在本章中，我们了解了如何创建和配置一个 DynamoDB 表。你已经在我们应用中创建了一些功能，可以从表中写入和读取数据，并且优雅地处理了认证超时。现在让我们用第 1 章中用过的`./sspa deploy_bucket`命令来部署应用。完成之后，可以在浏览器中启动应用进行测试，同时考虑下一步要做什么。

下一步

既然理解了 DynamoDB 的工作原理，下面这些延伸话题你可能会感兴趣。

jQuery 3 和 Promise/A+ Deferred

我们可以轻松地修改本章中看到的一些功能来使用 A+ Promises，用 jQuery 3 或者 ES6 均可。你可能需要修改一些函数名，这取决于你选择用 jQuery 3 还是 ES6。不过，整体的效果应该是相同的。将 AWS `DocumentClient` 的 API 封装在一个 Promise 中是轻松绑定异步事件的好方法，而且更易于让 DynamoDB 和应用的其他部分集成到一起。

策略模拟器

尽管所有这些属性配置方法都很酷、很全，但除非能对它进行测试，否则并不能给你带来任何好处。要保证不让有些人具备对某些东西的访问权限，是一件非常棘手的事情。幸运的是，使用 IAM 策略模拟器[12]，你可以轻松检查自己是否正确创建了一个访问配置（access profile）。注意，必须登录到 AWS 控制台才能使用这个工具。

12 https://policysim.aws.amazon.com/home/index.jsp

允许 S3 访问

并不是所有情况下把数据放进数据库都是合理的。像图像、声音和视频这些最好存储在专门为大型多媒体对象设计的服务里。这里你可以使用 S3，还可以使用我们在 DynamoDB 中用过的技术来控制 S3 对象的访问。查看 AWS 文档[13]，能获得更多信息。

13 http://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_examples.html#iam-policy-example-s3

第 6 章

使用Lambda构建微服务

尽管我们更倾向于在浏览器中执行应用所需的所有代码，但这种方法有时候是行不通的。在有些情况下，创建一个自定义 Web 服务来支撑应用会更合理。正如我们在第 5 章看到的那样，你可能需要限制数据访问，但又不能用访问策略实现。你可能还想将查询结果返回给浏览器之前对其进行过滤，为手机节省流量，抑或在受限的或者卡顿的网络条件下改善一点用户体验。给数据新增一个 HTTP 接口，就能让第三方访问你的应用或者通过浏览器原生的 HTTP 缓存策略来实现数据缓存。

我们要用 Web 服务，不意味着就要用到 Web 服务器。使用 Amazon 的 Lambda 服务，我们可以创建运行在专有容器中的服务，并且只按照实际发生的请求处理付费。与 EC2（Elastic Compute Cloud）和其他虚拟服务器环境不一样，用 Lambda 来构建自定义 Web 服务具有低成本、可扩展性和可用性等优点，就像我们使用的 DynamoDB 服务一样。正如本章将会看到的，Lambda 服务很容易构建和维护，而且你可以结合 AWS 生态系统的其他功能实现一些奇妙的东西，

例如，在我们的应用里，可以构建一个 Lambda 服务来为一个编程问题提供最通用的答案。我们可以通过安全的方式实现，而无须允许访问其他用户数据，并且把它实现为一个自定义的服务还能减少发送给浏览器的数据量。

为了搭建我们的服务，首先了解一点 Lambda 环境的基本知识以及它的工作原理。然后我们会用 AWS 命令行接口定义一个服务，编写代码，然后部署。最后，我们将会研究一下触发这个新服务的两种不同方式：用 Cognito 认证的授权方式；使用 HTTP 的公开服务方式。

理解 Amazon Lambda

Amazon Lambda 是一个能让你在云端运行代码，而不用维护和管理服务器的服务。有了 Lambda，Amazon 坚定地跳上了微服务的马车，甚至在拉动这驾车。Lambda 允许你用 JavaScript 编写独立的函数，甚至能用其他语言编写这些函数，然后在指定容器中运行。你可以通过多种方式访问这些函数，包括 Amazon API 网关的 Web 方式。

首先，我们需要创建一个带有两个输入参数的 JavaScript 函数，把它写入一个 `.js` 文件中。这个函数可以使用 Node.js 标准库或者我们想用的其他 Node.js 三方库中的任何方法。然后我们需要将函数和它的所有依赖打包成 zip 文件。最后，要选择一个名字和所有可配置参数，比如资源限制，再将包含我们的函数的 zip 文件上传到 AWS。至此，我们的新服务就可以使用了，不论通过我们所配置的哪种访问方式都行。

为了给应用构建一个新服务，首先让我们了解一下使用 Lambda 服务时所用的运行环境。然后我们将看看 Lambda 的定价。和所有的 Amazon Web 服务一样，在设计应用时，服务的成本影响了最终决策，这对 Lambda 也适用。了解了成本之后，我们将学习如何部署和构建一个新的 Lambda 服务。

Lambda 环境

针对这个应用，我们将使用 Node.js 来编写 Lambda 服务。Node.js 是一个基于 V8 引擎的 JavaScript 运行时，这个引擎用于 Google 的 Chrome Web 浏览器。类似 Ruby 的 EventMachine 和 Python 的 Twisted 框架，Node.js 使用了一个事件驱动模型来防止 I/O 阻塞

和其他长时操作。但与其他框架不一样，Node.js 标准库和开源库社区完全专注于这种事件驱动的方法，这样你的任何操作都可以用非阻塞的方式执行。

除了 Node.js，我们还有其他编写 Lambda 服务的方法。Amazon 也支持 Java 和 Python 的 Lambda 函数，所以如果你熟悉这些语言，也可以很顺手地使用这些编程环境。因为本书是关于构建 Web 应用的，所以我们将选择使用 Node.js，因为从事 Web 开发的人都会比较习惯使用 JavaScript。

为什么使用 Node.js?

Node.js 这几年非常流行，但并非没有争议。反对者批评有时候异步编程带来反直觉的体验，以及质疑为什么有人会在 JavaScript 之上构建新的运行时，毕竟因为历史原因，JavaScript 有很多特性。支持者则推崇 NPM（Node.js 包管理器）的包仓库，Node.js 基金会号称其为“世界上最大的开源库生态系统”（https://nodejs.org/en/），并称赞 JavaScript 容易理解，普及度高以及有很多学习资料。

你当然不一定非得用 JavaScript 编写 Lambda 服务，毕竟 Amazon 还提供了 Java 和 Python 两种选择。老实说，单就使用语言本身而言，JavaScript 在这三者中也是我最不愿意选择的。但 JavaScript 是一种需要被认可的语言，我们已经在使用它开发我们的应用。

不管你使用哪种编程语言，所有的 Lambda 服务都运行在基于 Amazon 的 Linux 发行版的服务器上，目前使用的是 3.14 内核。如果有兴趣，你可以查看 AWS 文档[1]找到对应的 Linux 版本号。从你的 Lambda 服务访问底层的 Linux 不仅仅是被允许的，更是被鼓励的。它是一种通过运行原生代码扩展 Lambda 能力的方式。

Lambda 服务入口是一个函数。这个函数的签名（signature）需要两个参数。第一个参数是 JSON 格式的请求有效载荷（类似于 HTTP post 的正文），以解析的 JavaScript 对象形式传给该函数。第二个参数被称为上下文，它是用来和 Lambda 环境交互的，包括用于响应请求的方法以及在发生错误而失败后的处理方式。以下是一个简单的 Lambda 函数例子，

1 http://docs.aws.amazon.com/lambda/latest/dg/current-supported-versions.html

你可以在预备的工作空间中找到它：

learnjs/1000/services/lib/index.js

```
exports.echo = function(json, context) {
  context.succeed(["Hello from the cloud! You sent " + JSON.stringify(json)]);
};
```

这个函数直接调用 succeed 函数完成请求响应，并返回参数值作为结果。Lambda 函数的结果都是 JSON 格式。JavaScript 的 Lambda 函数能产生一个 JSON 兼容的 JavaScript 对象，没必要调用 JSON.stringify()，因此我们在这里使用一个数组来保存消息。

用 Java 编写的 Lambda 函数在被频繁调用时，会因为 JVM 启动时间而损失性能。

有一点需要重视的是，Lambda 函数在返回时并不一定会退出。你的函数会一直消耗计算时间，直到你使用上下文响应请求或者返回失败。这是一个很重要的特征，因为你在运行异步任务，它们在函数返回时不一定会结束。在 Node.js 提供的异步环境中编程时，调用回调函数来完成请求响应会更合理。我们将在本章的后面创建一个 Lambda 函数，仔细研究它是如何工作的。

如果你没有完成请求，但也没有任何处于等待中的回调函数，你的函数将会退出，并显示消息“进程在请求响应前退出”。即使你在函数中不需要返回一个响应，这种情况仍然需要避免，因为你很可能占用了比实际所需更多的计算时间，以及在日志中产生不必要的错误信息。

Lambda 的局限

正如你可能想到的那样，在 Lambda 函数中有些事情是无法完成的。然而，仍有相当多的事情可以做，甚至你可能不会想到它们能在 Lambda 这样的环境中实现。

例如，你可以往底层操作系统的文件系统写入数据。每个 Lambda 函数在文件系统的 /tmp 目录下有 512 MB 临时磁盘空间，可以用来当作暂存空间（scratch space）。然而，

这个空间仅当你的函数在运行时才可使用。它不能在不同执行环境中持久化或者共享。

你还可以在函数中使用 shell 来运行 Linux 命令。这个功能使你能使用带原生扩展的 Node.js 库，或者使用标准 Linux 命令行工具来处理数据。你甚至可以上传自己的静态链接 Linux 可执行文件，然后从函数中调用它们。Lambda 的安全模型没有对函数允许做的事情和产生的子进程可以做的事情做区分。

在大部分 Linux 环境中，对可以产生的子进程数量和可以打开的文件描述符有一些限制[2]。它还限制了在并发执行时一次可以调用多少个函数。默认情况下，这个数字被设置为 100，但你可以联系 Amazon 客服增加这个值。最后，无论在 Lambda 函数中定义了什么设置参数，单次执行不能超过 300 秒。诸如此类的限制是为了防止程序错误演变成一次昂贵的计算错误。为了理解这些错误的代价可能有多高，我们先看一下 Lambda 的服务是怎样计费的。

内存、时间和费用

只要与 DynamoDB 一起，Amazon Lambda 就能提供近乎无限的可伸缩性和低成本的高可用性。用它来构建我们的 Web 服务，意味着我们的应用可以变得非常流行，而无须担心为了扩展而进行重构的问题。但为了更有效地使用这些服务，你必须理解它们是如何收费的。否则，你的低成本解决方案可能快速演变成意料之外的账单灾难。

使用 Lambda 时，按照千兆秒（gigabyte-second，GBS）付费。你的函数运行得越久，使用的内存就越多，要付的费用也越多。例如，假设你的函数需要 128 MB 的内存，运行时间为 100 ms。如果它被调用 100,000 次，你需要为 125 GBS 支付相应费用（根据当前定价，金额为 0.02 美元）。

所以，在写 Lambda 函数时需要注意的这两个成本就是我们写所有程序时都需要考虑的两个成本：空间（内存）和时间。这包括以 shell 方式在环境中执行的其他进程所使用的

2 https://docs.aws.amazon.com/lambda/latest/dg/limits.html

内存，不管这些进程是包含在 Amazon Linux 发行版中还是作为函数的一部分上传的。

你的 Lambda 函数执行的时长是指从它开始执行到退出的时间。它不是基于使用了多少 CPU 时钟周期来计算的。并行发送异步请求而不是连续发送，不仅保证你的函数顺利地运行，还能够保证低成本。此外，利用 Lambda 函数的暂存磁盘空间是节省内存（但是，可能会消耗更多时间）的好方法。在硬盘中积累结果可能比把它们保留在内存中更合理。

既然我们讲解了 Lambda 能做什么，不能做什么，以及收费情况，是时候开始构建一些东西了。就我们的 Web 应用而言，我们将通过部署一些东西来创建一个新服务。然后，专门为应用构建一个新服务。

先部署

正如我们在构建 Web 应用前部署它一样，出于同样的原因，我们将在构建一个 Lambda 服务前先进行部署。通过这个过程，你将会看到如何对服务进行配置、部署和测试。我们没有开发一个新服务，而是将部署一个已有的服务。一旦完成了这一步，就可以构建一个新服务，并且自信十足地部署它。

我们将要部署的这个服务是基于本章前面的 `echo` 函数的。在预备的工作空间里，`services` 目录包含了我们应用的 Lambda 服务的所有代码，`echo` 函数的代码也在这个文件夹内。不仅是代码，预备的工作空间中还包含了控制如何配置单个微服务的文件。正如我们应用中的其他服务一样，这些配置文件存放在 `conf` 文件夹下。这两个工件合在一起，我们部署一个自定义 Web 服务所需的一切就备齐了。

正如其他任务一样，你可以用 `sspa` 脚本来运行必要的 AWS CLI 命令部署 Lambda 函数。我们会一步一步地解释，以便你了解是怎么回事。要部署这个服务，请先检查包含在预备工作空间中的 Lambda 样例配置。然后我们会看一看这个 echo 服务是如何部署的，并且测试它的配置是否正确。

配置一个 Lambda 函数

首先，打开 conf/lambda/functions/echo 目录下的 config.json 文件。

learnjs/1000/conf/lambda/functions/echo/config.json

```
{
    "Runtime": "nodejs",
    "Timeout": 5,
    "MemorySize": 128,
    "Publish": true
}
```

这个配置文件被传给 AWS CLI 的 lambda create-function 命令[3]。这里的 Runtime 属性指定了函数使用的运行时环境，可选项包括 nodejs、java8 和 python2.7。我们的服务使用 Node.js，所以需要把这个属性值设置为 nodejs。

如前所述，Lambda 函数是根据它们使用的内存数和运行时间来收费的。Timeout 和 MemorySize 属性限制了函数能使用的内存数和时长。echo 函数的配置是，使用的内存不超过 128 MB，运行时间不超过 5 秒。考虑到它做的事情，这个配置足够了。对于新函数，我们可以先选择一个合理的设置，如果不够，再提高这两个值。

Publish 用于设定，当我们发布新版函数时是否创建一个唯一的版本号。你可以使用这些版本号来引用 Lambda 函数，以确保其行为不会出乎意料地改变。尽管我们目前不会使用 Lambda 函数版本号，但是仍然在新服务中将这个参数设置为 true，以免之后改变主意。

Lambda 函数的加载时间

为了更加简洁，这个应用里所有函数句柄（function handler）都写在文件 index.js 中。然而，随着应用的演进，你可能希望把每个函数句柄移出来，放到它们自己的文件里。把每个服务各自的依赖分开，意味着该服务启动时需要加载的模块更少，这样能改善服务响应的次数。我们应用的服务中没有太多依赖，所以目前我们还不需要这么做。

3 http://docs.aws.amazon.com/cli/latest/reference/lambda/create-function.html

除了 config.json 文件中的属性，sspa 脚本自动设置了 lambda create-function 命令所需的其他属性。比如，FunctionName 属性被设成配置目录的名字。Role 属性被当作 IAM 角色中的 ARN。sspa 脚本在 AWS CLI 中使用 lam createrole 命令生成了这种角色。这个角色的配置如下：

learnjs/6000/conf/iam/roles/learnjs_lambda_exec/info.json

```
{
    "Role": {
        "AssumeRolePolicyDocument": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Action": "sts:AssumeRole",
                    "Principal": {
                        "Service": "lambda.amazonaws.com"
                    },
                    "Effect": "Allow",
                    "Sid": ""
                }
            ]
        },
        "RoleId": "AROAJ67KFEW5PCLEK3X2S",
        "CreateDate": "2015-12-17T18:27:47.499Z",
        "RoleName": "learnjs_lambda_exec",
        "Path": "/",
        "Arn": "arn:aws:iam::730171000947:role/learnjs_lambda_exec"
    }
}
```

在第 5 章，我们使用一个 IAM 角色来管理用户的权限。这里的角色不如那个复杂，因为它唯一的限制条件是，它应该是一个 Lambda 服务。现在这个角色没有 profile（配置文件），因而也没有特殊访问权限。我们需要给这个角色添加 profile 来支持新服务，但对于 echo 函数而言，不需要任何新的权限。

Handler 属性定义了函数句柄（function handler）。函数句柄是指当 Lambda 函数被调用时实际执行的 JavaScript 函数的名字，它是通过对应的 Node.js 模块（文件）以及导出

的函数名来指定的。sspa 脚本假定这个函数名和配置目录名是一样的，而且所有的 Lambda 函数都存放在 servers/lib 文件夹下的 index.js 文件中。比如，echo 函数的句柄就是 index.echo。

创建代码包

我们需要把 Lambda 服务的代码打包成一个 zip 文件，里面包含函数和它们的依赖。这个 zip 文件称为代码包（code bundle）。你需要创建这个代码包才能为 echo 函数上传配置。

代码包大小不能超过 50 MB。

为了构建服务的代码包，你需要先安装 Node.js。Amazon 的 Lambda 环境目前使用 Node.js 4.3.2 版本，而且要使用同样的版本来运行和测试 Lambda 函数。另外，你还需要 NPM 包管理器来安装依赖。如果要安装这些工具，可以参考附录 A。

一旦安装了 Node.js 和 NPM，就可以通过构建服务代码包来测试你的设置。sspa 脚本在 services 文件夹下生成名为 archive.zip 的 zip 文件，就创建了这个包。你可以在脚本中使用 build_bundle 动作来构建代码包，比如：

```
learnjs $ ./sspa build_bundle
```

执行成功后，就可以把 echo 函数上传到 AWS。做这件事之前，先在 sspa 脚本中执行 create_service 动作：

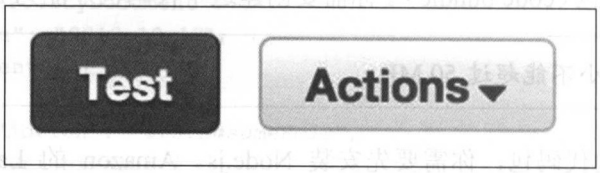
```
learnjs $ ./sspa create_service conf/lambda/functions/echo
```

如果这条命令执行没有出错，这个 echo 服务应该就可以使用了。我们应该试用一下，对吧？当然，对这个服务我们没有访问端口，也不打算花时间来建一个，但还是有方法来测试它的。为了进行测试，我们返回 AWS 的 Web 控制台，打开 Lambda 服务的控件。

通过 AWS 控制台测试函数

为了测试函数的实际逻辑，我们将用 Jasmine 来编写单元测试，就像之前测试运行在浏览器中的应用代码一样。我们需要用不同的方式来测试刚才创建的服务配置。一种方法就是使用 AWS 控制台自带的测试功能。

如果你看一下 Lambda 函数的设置页面，应该可以在左上角看到一个名为“Test”的按钮，如下图所示。



单击这个按钮，就会出现一张表单让你填写请求主体。发送给 Lambda 的请求数据都是 JSON 格式，所以无论这里输入什么内容都必须是有效的 JSON。不过，因为我们的函数只是简单地返回发送的内容，JSON 消息的格式并不重要。

一旦输入一个值，单击“Save”和“Test”按钮，函数就会被调用。AWS 控制台在屏幕的底部显示结果，它还会显示函数是成功返回还是出错了。如果一切都成功运行，你应该可以看到如下图所示的结果。

```
[
  "Hello from the cloud! You sent {\"problemNumber\":42}"
]
```

如果看到这样的结果，就知道 `echo` 函数已经被正确部署了。很明显，这只是一个冒烟测试（smoke test）。也就是说，这个测试告诉我们该函数可以运行，但并不会告诉我们它是否在所有情况下都能正常工作。这是一个简单的函数，但随着我们创建的函数越来越复杂，将需要一个方法对它们进行更加完整的测试。

创建一个新的 Lambda 配置

既然为 Lambda 服务 echo 部署了配置，现在我们为新服务创建一个新配置。然后我们可以写这个服务，并上传代码。要为我们的新函数创建配置，直接复制 echo 目录并命名为 popularAnswers。一定要在复制后删除 popularAnswers 目录的 info.json 文件，这样 sspa 脚本就知道要创建一个新服务。然后你就可以在新目录中按需修改 config.json 文件了。

实际上，echo 服务的配置和我们新服务需要的配置已经非常接近了。我们可能需要做的一件事情是将最大内存限制从 128 MB 提升至 512 MB。你可以通过修改 MemorySize 属性来实现，代码如下：

learnjs/6001/conf/lambda/functions/popularAnswers/config.json

```
{
    "Runtime": "nodejs",
    "Timeout": 5,
    "MemorySize": 512,
    "Publish": true
}
```

一旦修改完毕，就可以运行 sspa 脚本，将这个新配置上传到 AWS：

```
learnjs $ ./sspa create_service conf/lambda/functions/popularAnswers
```

如果运行没有问题，我们的新服务就配置成功了。它暂时还没有任何代码，所以没必要做测试，但我们在下一节中会补上。

创建一个测试环境

尽管在这里我没有写，但你可能会想要创建两个函数而不是一个。就我们应用余下的部分而言，为这个 Lambda 函数创建一个测试环境，可以在它们被部署给用户之前确认修改，它将为你提供一个安全的开发沙盒。

往 Lambda 执行角色上添加策略

既然创建了新的服务，就需要添加一些策略到这个 IAM 角色上，Lambda 服务在运行时会担任这个角色。这个由 sspa 脚本创建的名为 learnjs_lambda_exec 的角色，需要一个策略来允许我们的 Lambda 服务访问 DynamoDB。因为你可能希望为每个服务单独配置它们的角色，我们打算尽可能简化，让应用的所有的服务使用同一角色。这意味着我们的所有服务都有相同的访问权限。

我们将使用 Amazon 的托管策略，而不是为 Lambda 服务角色创建自定义策略。这些都是预定义的策略（canned policy）——由 Amazon 创建并管理——它提供了对所有资源的访问权限，不管是一个特定的服务还是一个范围的服务。你可以在管理员权限下执行 iam list-policies 命令以查看所有可用的托管策略：

```
$ aws --profile admin iam list-policies
{
    "Policies": [
        {
            "PolicyName": "AWSDirectConnectReadOnlyAccess",
            "CreateDate": "2015-02-06T18:40:08Z",
            "AttachmentCount": 0,
            "IsAttachable": true,
            "PolicyId": "ANPAI23HZ27SI6FQMGNQ2",
            "DefaultVersionId": "v1",
            "Path": "/",
            ...
```

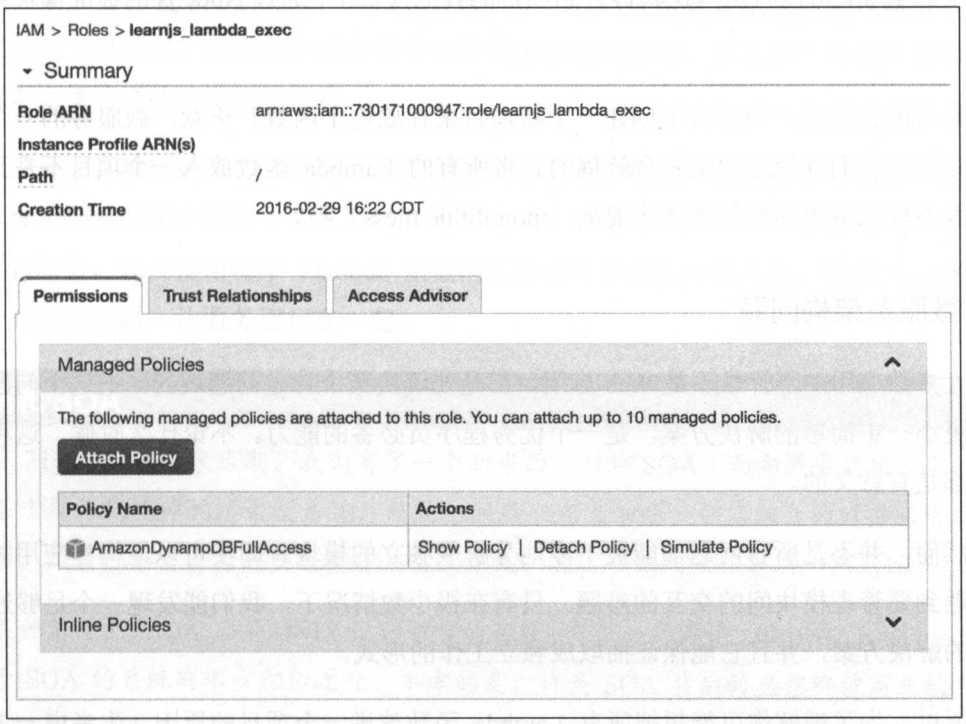
我们希望加到 Lambda 服务上的策略名为 AmazonDynamoDBFullAccess。我们需要策略的 ARN 把它添加到 Lambda 执行角色上。托管策略的 ARN 仅仅是基于名字的。这里的就是 arn:aws:iarm::aws:policy/AmazonDynamoDBFullAccess。

为了把这个策略添加到 Lambda 执行角色上，需要在 AWS CLI 中运行 iam attach-role-policy 命令。可以使用 sspa 命令来执行，但与本章中执行的其他任务不一样，用 AWS CLI 会更简单。只要有角色名和策略的 ARN 即可，如下所示：

```
$ aws --profile admin iam attach-role-policy \
```

```
--role-name learnjs_lambda_exec \
--policy-arn arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess
```

一旦这个命令执行完成——假设你没有收到任何错误——我们的所有 Lambda 服务对 DynamoDB 就都具备了完全访问权限。由于我们还没有为这个新服务增加一个实现，所以你可以通过 AWS CLI 中的 `iam list-attached-role-policies` 命令或者查看 AWS Web 控制台的角色部分来验证。应该可以看到一个添加上的托管策略，如下图所示。



现在我们配置了一个新 Lambda 服务，可以开始编写实际支撑服务的 JavaScript 函数了。我们的新函数将需要从请求中获取一个题目编号，然后从数据库中取出该题的所有答案。它还需要用一个比较器对这些答案进行排序，找到最受欢迎的那些，然后把这些答案作为响应返回。

编写 Lambda 函数

为了给我们的新服务编写函数，我们打算向 `index.js` 文件再添加一个函数定义。此函数名为 `popularAnswers`。一旦部署了这个函数，就能使用我们创建的这个新服务。

你想加到这个应用的任何其他 Lambda 函数都应该被加到 `index.js` 文件中。随着函数数量的增加，可以将这些逻辑拆分到项目的其他模块中，通过 Node.js 的标准模块系统引入[4]。

你可能会好奇，为什么不创建一个新项目来存放这个函数？毕竟，微服务的一个明显的好处就是它们互相之间是完全解耦的。将所有的 Lambda 函数放入一个项目不是又制造了微服务理应避免的那种单体式混乱（monolithic mess）吗？

规避微服务架构问题

大多数应用，不管是不是 Web 应用，都是要解决某个特定问题域的。把这个问题域分解成更小、更简单的解决方案，是一个优秀程序员必备的能力。不论什么时候，这种问题分解都是有意义的。

然而，并不是所有问题都能被干净地分解成独立的模块。即使可以，构建应用时也常常会遇到要管理模块间的交互的难题。只有在极少数情况下，我们能发现一个足够通用和解耦的解决方案，并且它能保证抽取成独立工作的形式。

所以，为了理解你可能想把所有 Lambda 函数放进一个项目的原因，先考虑一下如果不这么做会发生什么。如果很难将应用的各部分干净利落地解耦，那么用解耦型的结构就是一种极端傲慢的行为。你很可能要多次重构设计才能得到一个解耦、完整封装、可重用的独立的服务。

如果一开始就对应用进行拆解，过早优化，那么之后你将会发现需要跨服务重构来优

4 https://nodejs.org/api/modules.html#modules_modules

化整体设计，工作量很大。在单体系统中原本做一个修改就好了，这下子可能变成三个。对一个代码库的两次提交可能会演变成对多个代码库的五六次提交。这种情况是可能发生的：一开始就把服务分开反而再也得不到真正完全解耦的服务，因为这种做法让通过重构来改善设计变得更困难了。

为应用服务创建一个独立的项目，让我们能在服务间轻松地共享代码，重用测试数据和辅助函数来验证服务间的交互，以及通过维护单一代码源来杜绝重复的领域模型。如果在将来的某个时间点，抽取某个或者多个独立服务会更为合理，那么让代码保持在容易重构的状态会使这样的转变更加容易。

如果说设计糟糕的单体系统是一个大泥球，那么设计糟糕的微服务系统就是一大群红蚂蚁，你不会希望某个周日午后在后院里看到这堆乱糟糟的东西。通过部署同一个代码包里的所有服务，以及使用每个 Lambda 函数的配置来控制执行什么代码，我们可以使用一个很好的服务容器，让服务运行在一起。

微服务和 SOA

在 21 世纪 90 年代后期，我们有了一个新东西，叫作 SOA（面向服务的架构）。它不是基于单体代码库构建大型系统，而是把问题一点点细分，创建独立的服务来管理各种需求，然后根据需要为这些服务建立交互方式，通过一系列服务解决一系列问题。

某种意义上，SOA 只是 UNIX 设计哲学的扩展：只做一件事情，做好它。bash pipeline 和基于 SOA 的系统有不少相似之处。不幸的是，许多 SOA 背后的思想都被商业软件公司粗暴地曲解，然后我们发现自己开始谈论服务总线、服务发现、SOAP、WSDL 以及很多其他可怕的东西。其实我们应该集中精力解决的是，如何让这些服务更简单、更易于访问，而不是用花哨的方式来组织和连接它们。

18 世纪的外交官 Charles Maurice de Talleyrand 说过，“政治家的一个重要手段就是当机构对公众而言变得面目可憎时，为机构换一个新名字。”[5] 微服务不过是把 SOA 重

5 http://www.azquotes.com/author/14429-Charles_Maurice_de_Talleyrand

新推荐给新一代程序员，因为他们还未曾被过去的那些错误吓破胆，我真的很难不这么想。也许这一次，我们能找到正确的路。

添加服务依赖

因为你在构建自己的服务，所以很可能需要添加一些库来支持它们。你需要的所有依赖都必须包含在代码包中，放在标准的 Node.js 包目录 node_modules 下。为了在我们的服务中安装依赖，要使用 Node.js 包管理器 NPM。

我们需要的第一个依赖是 AWS JavaScript SDK。它应该已经包含在预备的工作空间内。只需要检查 package.json 文件的 dependencies 部分，就可以轻松确认。代码如下所示：

learnjs/6000/services/package.json

```
{
  "name": "learnjs",
  "version": "1.0.0",
  "description": "learnjs",
  "main": "index.js",
  "scripts": {
    "test": "make test"
  },
  "author": "Ben Rady",
  "license": "All Rights Reserved",
  "dependencies": {
    "aws-sdk": "2.2.4"
  },
  "devDependencies": {
    "jasmine": "2.3.1"
  }
}
```

再次执行 sspa build_bundle 命令，确保你可以安装这些依赖。这条命令会把依赖安装到 services/node_modules 目录下。构建部署包时，这个目录下的所有依赖都被包含在内，你全部的函数都能访问。既然依赖问题已经解决，就可以开始编写测试了。

构建可测试的服务

和应用中所有的代码一样，我们的新 Lambda 函数也从测试开始。为了给这个函数编写测试，我们将使用与单页应用相同的测试框架。不过，你会看到我们将需要在一个完全不同的环境下执行测试。

Jasmine 测试框架既能在浏览器中运行，也能在 Node.js 这样的后端环境中运行。目前，工作空间中有一个测试可以测试示例函数 echo。使用 sspa 命令，你可以执行所有测试，代码如下：

```
learnjs $ ./sspa test
Started
.

1 spec, 0 failures
Finished in 0.005 seconds
```

现在，我们的应用服务只有一个 echo 函数和一个测试。我们想要添加一个新的函数和一个 describe 节来填充它的测试代码。当服务变得越来越臃肿时，你会想要把所有东西分离到独立的文件中，但是就目前而言，我们可以把测试代码放进 index_spec.js，把程序代码放进 index.js。让我们看看这个新函数（带上下文的）的一些测试代码：

learnjs/6302/services/spec/index_spec.js

```
describe('lambda function', function() {
  var index = require('index');
  var context;

  beforeEach(function() {
    context = jasmine.createSpyObj('context', ['succeed', 'fail']); index.dynamodb
    = jasmine.createSpyObj('dynamo', ['scan']);
  });

  describe('popularAnswers', function() {
    it('requests problems with the given problem number', function() {
      index.popularAnswers({problemNumber: 42}, context);
      expect(index.dynamodb.scan).toHaveBeenCalledWith({
        FilterExpression: "problemId = :problemId",
```

```
      ExpressionAttributeValues: { ":problemId": 42 },
      TableName: 'learnjs'
    }, jasmine.any(Function));
  });
  it('groups answers by minified code', function() {
    index.popularAnswers({problemNumber: 1}, context);
    index.dynamodb.scan.calls.first().args[1](undefined, {Items: [
      {answer: "true"},
      {answer: "true"},
      {answer: "true"},
      {answer: "!false"},
      {answer: "!false"},
    ]});
    expect(context.succeed).toHaveBeenCalledWith({"true": 3, "!false": 2});
  });
 });
});
```

正如你在之前测试中所见，可以用 Jasmine 的 spy 对象掐灭和 DynamoDB 的交互。你可以写几个关于对象的方法如何被调用的断言，来测试服务是否使用了正确的查询参数。通过检查对 scan 方法的调用，可以触发传入的回调函数并模拟一个响应。

了解 API 的工作机制后再开始构建它

有一个方法能快速查看真正的 DynamoDB API 在这种情况下究竟返回了什么：不要马上伪造交互。JavaScript AWS 库使用主目录中（~/.aws/credentials）的默认证书来真正发送请求。当请求返回时，你就可以在测试输出中看到真实的响应结果。

记住，这个请求是异步的，你很可能需要用到 Jasmine 支持的异步测试，就像我们在第 4 章中做的一样。另外，还要记住，你可能有执行破坏性操作的权限（取决于默认的证书属于哪个用户），所以在使用服务时要多加小心。知道这个 API 的工作原理后，你可能希望伪造那个交互，让你的测试快速和可靠。

调用 Lambda 函数时可以创建一个假的上下文。写几条关于这个对象的哪个方法被调用和如何被调用的断言，能帮助确保你的服务是正常工作的。你不仅可以使用这个对象来

测试 Lambda 函数的预期行为，还可以测试它是如何处理失败情况的。

查询、分组和分页

为了实现我们的服务函数，需要根据 item 的 `answer` 属性对其进行分组。DynamoDB 没有内置的方式来做分组操作，但我们可以在自定义服务（有一些限制，后面会看到）中实现。这些测试断言我们在执行 DynamoDB 查询请求并且会返回正确的结果。如果发生错误，我们调用 `context.fail` 方法来确保函数退出和在日志中记录下错误。

这里，你可以看到这个满足上一节测试的新服务函数的一种实现。没有必要在你的应用中也使用这种实现。事实上，你可以在它的基础上做一点改进（我们之后会讨论），但这个实现能帮助你了解该服务应该做些什么。

learnjs/6302/services/lib/index.js

```
var http = require('http');
var AWS = require('aws-sdk');
AWS.config.region = 'us-east-1';
var config = {
  dynamoTableName: 'learnjs',
};
exports.dynamodb = new AWS.DynamoDB.DocumentClient();
function reduceItems(memo, items) {
  items.forEach(function(item) {
    memo[item.answer] = (memo[item.answer] || 0) + 1;
  });
  return memo;
}
function byCount(e1, e2) {
  return e2[0] - e1[0];
}
function filterItems(items) {
  var values = [];
  for (i in items) {
    values.push([items[i], i]);
  }
var topFive = {};
values.sort(byCount).slice(0,5).forEach(function(e) {
```

```
      topFive[e[1]] = e[0];
    })
  return topFive;
}
exports.popularAnswers = function(json, context) {
  exports.dynamodb.scan({
    FilterExpression: "problemId = :problemId",
    ExpressionAttributeValues: {
      ":problemId": json.problemNumber
    },
    TableName: config.dynamoTableName
  }, function(err, data) {
    if (err) {
      context.fail(err);
    } else {
      context.succeed(filterItems(reduceItems({}, data.Items)));
    }
  });
};
exports.echo = function(json, context) {
  context.succeed(["Hello from the cloud! You sent " + JSON.stringify(json)]);
};
```

这种方法适用于小结果集，但有点简单。为了扩展它，我们需要对扫描结果进行分页。一次查询或者扫描请求的结果数据不能超过 1 MB。这意味着如果应用中有超过 1 MB 的答案，我们创建的这个服务就看不到全部答案内容。

DynamoDB 响应包含了 `LastEvaluatedKey` 字段来允许我们对大于 1 MB 的结果[6]进行分页。把 `ExclusiveStartKey` 属性值设置为与 `LastEvaluatedKey` 相等，然后重新提交请求，我们可以按照 1 MB 为增量对结果进行分页。

这样我们的微服务在不超过 DynamoDB 响应大小限制的情况下，将所有最受欢迎的答案集中到一起。它还能让我们微服务的内存使用量保持在较低水平，以节省费用。

为了部署这个服务，你需要使用 `sspa` 脚本执行 `deploy_bundle` 操作，如下所示：

---

6 http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/QueryAndScan.html#Pagination

```
learnjs $ ./sspa deploy_bundle
```

这条命令运行了你之前运行过的 `build_bundle` 命令，然后取出生成的代码包，在 AWS CLI 中使用 `lambda update-function-code` 命令更新应用的所有函数。

一旦部署了代码，就可以从我们的应用中调用这个新服务了。我们并不会像第 3 章一样，带你走一遍构建服务的流程，我们将直接跳到核心部分——创建 Web 服务调用。

调用 Lambda 函数

可以通过两种方式在浏览器中调用 Lambda 函数：第一种是使用 AWS SDK；第二种是通过 Amazon API 网关调用。首先，我们来看如何使用 AWS SDK 和颁发给用户的 Cognito 证书从浏览器调用 Lambda 函数。

为已认证的 Cognito 角色添加新策略，就能从浏览器直接调用 Lambda 函数，无须使用公共的 HTTP 界面。只要用户有符合该角色的合适证书，他们就可以执行策略允许的任何操作。我们想执行的一个操作名为 `invoke`。为了执行这个操作，你需要从 AWS 库中创建 `Lambda` 类的实例，然后需要调用 `invoke` 函数。

创建一个 Lambda 策略

为了允许针对这个 Lambda 函数的访问，你需要创建一个新的 IAM 策略，然后把它添加到授权的 Cognito 角色上，就像我们在第 5 章的“授权访问 DynamoDB”一节中做的那样。与那个策略不同的是，这个策略中不需要 Condition 子句，因为你为所有授权用户都授予了访问权限。

当然，在使用 Lambda API 时，会有和 DynamoDB 一样的证书过期问题。如果我们能重用 `sendDbRequest` 函数来处理所有这类事情，是不是非常美好？事实上，我们完全可以这样做，但你可能希望给这个函数换个名字，比如 `sendAwsRequest`。如果换了名字，这些测试还是能通过，可以继续写一个函数来调用此服务：

learnjs/6200/public/app.js

```
learnjs.popularAnswers = function(problemId) {
  return learnjs.identity.then(function() {
    var lambda = new AWS.Lambda();
    var params = {
      FunctionName: 'learnjs_popularAnswers',
      Payload: JSON.stringify({problemNumber: problemId})
    };
    return learnjs.sendAwsRequest(lambda.invoke(params), function() {
      return learnjs.popularAnswers(problemId);
    });
  });
}
```

这个新函数的测试与 DynamoDB 函数的测试模式是相同的，因此很容易编写。正如第 5 章所示，我们使用 jQuery 的 `Deferred` 对象来协调这些请求。在我们应用的视图中添加这些信息要用到几乎相同的表单。

这种方法有一个限制：只有授权用户能够发送请求。虽然放开对这个函数的许可，允许任何人调用，也不是不可能，但还有一种方式提供对这个 Lambda 函数的公开访问权限，这样需要的人都可以调用它。接下来，我们将看一下如何通过 Amazon 的 HTTP API 网关提供对 Lambda 函数的访问。

使用 Amazon API 网关

正如我们看到的，通过带 Cognito 证书的 AWS SDK 调用 Lambda 函数，是把自定义服务集成到应用的好方法，但如果要提供 Lambda 函数的公开访问权限呢？可以用 Amazon API 网关通过一个未授权的 HTTP 请求访问这些函数。

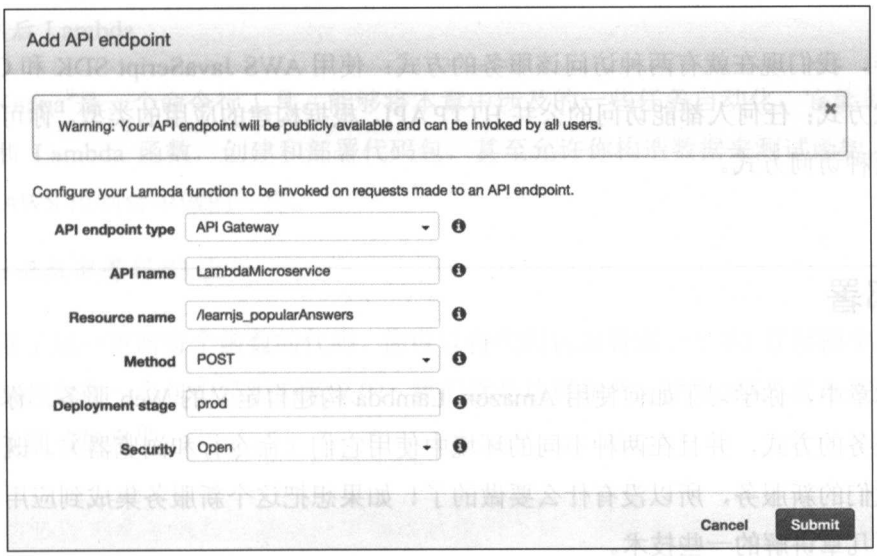
HTTP 还是 HTTP?

虽然 API 网关是“一种通过 HTTP 调用 Lambda 函数的方式”这种说法没错，但是需要理解一点，AWS SDK 中的 JavaScript 也是通过 HTTP 调用 Lambda 函数的。确实，

它与 AWS 几乎所有的交互都是通过 HTTP 完成的，因为对于 Web 浏览器来说 HTTP 是目前为止最稳定的协议。API 网关只不过使用了不同的终端而已。

Amazon API 网关通过每个函数定义的终端将 API 与 Lambda 函数对应。你可以通过 Amazon API 网关控制台或者 Lambda 控制台创建 API 和它们关联的终端。为了给我们的函数创建一个公共的 API，这里将使用 Lambda 控制台。

首先，在 Lambda 控制台中打开函数的设置页面。选择“API endpoint”标签页，点击“Add API endpoint”链接。一旦完成，将会提示你选择一个终端类型。选择 API 网关，你将看到如下图所示的配置内容。



将“Security”设置为“Open”，在改这个设置的时候，不用理会冒出来的吓人的警告信息。这里我们就是想让它公开化。接着，把“Method”项的设置改为“POST”，因为我们希望使用 HTTP post 来传递请求主体。根据你创建的 API 版本，将“Deployment stage”设置为“prod”或者“test”，然后选择一个 API 名称。

API 名称是用来在网关控制台中识别这个 API 的。如果你用同一个 AWS 账号创建了很

多不同应用，可以用这些名字来区分不同应用的不同 API。

保存这些改动之后，可能得等几分钟它们才能生效。然后你可以尝试在命令行中发送一个请求，确保 API 是在线并且可公开访问的。从 Lambda 控制台中获得终端 URL，然后使用 curl 命令，从命令行中发送一个 POST 请求，如下：

```
$ curl -d '{"problemNumber":1}' «endpoint_url»
{"true":2,"!false":1}
```

如果你收到一个这样的响应，就知道服务是正常的。如果你没有保存过任何答案，可能会得到一个空 JSON 对象。如果你得到响应消息说“缺少授权令牌”，请检查一下你的 URL。

这样，我们现在就有两种访问该服务的方式：使用 AWS JavaScript SDK 和 Cognito 证书的授权方式；任何人都能访问的公共 HTTP API。根据构建的应用的类型，你可以使用一种或者两种访问方式。

重新部署

在本章中，你学习了如何使用 Amazon Lambda 构建自定义的 Web 服务。你看到了两种访问服务的方式，并且在两种不同的环境中使用它们（命令行和浏览器）。既然你已经部署了我们的新服务，所以没有什么要做的了！如果想把这个新服务集成到应用中，可以尝试前面几章讲解的一些技术。

下一步

了解了 Lambda 的工作原理，下面这些延伸话题你可能会感兴趣。

Cognito 用户认证

创建一个 Cognito 用户池时，可以指定当用户登录或者注册时候调用哪些 Lambda

函数。在创建一个账号之前可以用这些函数验证用户数据，初始化数据库条目，或者发送一个自定义验证消息。用户池是 Cognito 新添加的功能，提供了比认证池更多的集成点。参考 Cognito 官方文档[7]，能了解更多细节。

API 网关

API 网关也是 AWS 新加的功能，而且它的功能还在增加。本章我们看到了它的一种使用方式，但还有很多其他方式。比如，除了 Lambda 服务，你还可以使用 Amazon EC2 作为访问终端。你也可以使用自定义的 Web 服务作为终端。关注这个服务[8]，看看你能否找到其他的使用方式。

Kappa，然后 Lambda

Kappa[9]是一个命令行工具，能够将本章中涉及的一些任务自动化。它能创建 IAM 角色和 Lambda 函数，创建和部署代码包，甚至允许你构造数据来测试函数，就像我们在 AWS 控制台中做的一样。

将 Lambda 函数部署到 S3 上

除了逐一更新每个函数的代码，你可以将代码包部署到一个 S3 存储桶中。将所有代码都包含在一个包中是很有意义的，我们就是这样做的。参考 AWS 官方文档，可获得更多细节信息[10]。

---

7 http://docs.aws.amazon.com/cognito/latest/

8 https://aws.amazon.com/api-gateway/

9 https://github.com/garnaat/kappa

10 https://blogs.aws.amazon.com/application-management/post/Tx3TPMTH0EVGA64/Automatically-Deploy-from-Amazon-S3-using-AWS-CodeDeploy

第 7 章

无服应用的安全

构建安全的软件系统不是一件容易的事。说它难的原因之一是你不知道自己什么时候做对了。相反，有些时候发现做没做错则相当简单：把服务放到互联网上，安心等待，并留意会发生什么。

本章我们先回顾一些保证 AWS 账户安全的基本原则，然后讨论几种可以用来攻击无服 Web 应用的常见攻击。你将会看到这些攻击是如何实施的，以及如何防范它们。希望理解这些攻击的细节后，你将可以看到自己应用的弱点，并找到保护它的安全的方法。

> 任何人都可以发明一个他自己都无法攻破的安全系统……如果有人交给你一个安全系统，然后说："我认为它是安全的。"你首先需要问的是："哥们，你确认自己没有吹牛吗？"展示一下你攻破过的系统，我才会相信你对这个系统安全性的断言是有点道理的。
>
> ——布鲁斯·施奈尔，密码学研究者、作家

保护你的 AWS 账号

保护应用的第一步是确保你的 AWS 账户是安全的。如果攻击者可以登录你的 AWS 账户然后关闭它们，那么世界上没有一个应用安全卫士是有效的。

事实上，如果你的 AWS 账号被攻破，不仅仅是一个应用会有风险。无意中让一名攻击者进入你的 AWS 账户可能会成为一个昂贵的错误。例如，获取你 AWS 账户权限的任何人都可以启用成百上千的 *g2.8xlarge* EC2 实例，然后用它们开挖比特币，直到被人发现。已经有很多起这样的案例，有人编写爬虫查看公开的 Github 账号，寻找 AWS 证书信息来做这种事情。所以，在我们讨论应用安全性之前，先了解一些 AWS 账户安全的基础知识。

禁用所有 root 访问密钥

如果你已经为自己的账户创建了 root 访问密钥，应该删除它们。你不可能像对待普通 IAM 用户的密钥那样管理这些密钥，而且如果你需要撤销密钥或者追溯地限制允许它们访问的服务，基本上没有什么方案可以选择。

如果你还没有创建过这种类型的访问密钥，那就不要创建。它们能做的任何事情你都可以通过带管理员权限的 IAM 用户来实现。使用 IAM 用户的好处是你可以随时限制这些权限而无须完全停用这些密钥。这意味着如果你已经部署了一个使用这些密钥的脚本、应用或者工具，发现自己授予的访问权限有点多，则可以轻松地逐渐进行调整。

管理用户配置

在第 1 章“创建一个带访问密钥的 AWS 用户”一节中，我们创建了一个带管理员权限的 AWS 用户。这个用户对你账户中的所有内容有完全的访问权限，这就使创建表和服务这种工作变得很简单。然而，这个级别的访问权限意味着，在管理这个用户的访问密钥时你可能要更加谨慎，以防止它们被误用。

一种保证这些证书安全的方法是使用不同配置来提供不同级别的访问权限。`sspa` 脚本

使用 admin 配置来确保这些证书不会被无意中以非预期的方式使用，比如在运行 Lambda 服务的测试时，和 AWS 的 JavaScript 版 SDK 进行交互会使用那些默认证书，除非你告诉它不要使用。

我们在第 1 章运行 `aws configure` 时添加的 `profile` 属性告诉 AWS CLI 去添加这个证书到管理员配置中。无论什么时候，`sspa` 脚本运行和 AWS 交互的命令，都会再次使用这个`--profile` 选项。

保护 AWS 证书

另一种保证证书安全的方式是把它们保存到安全的地方。默认情况下，AWS CLI 将它的配置文件写到你的 home 目录下，这些文件是未经过加密的。只要获取了你电脑的物理访问权限，就可以读取这些文件。这种事情很可能以你意料之外的方式发生。假设你把旧电脑给了一个朋友，后来他自己买了一台新的，把这台旧的卖了，那么你的硬盘最后不知道会落到谁手里。而如果你的密钥文件仍旧在那块硬盘中，你可能会面对一张巨额 AWS 账单。

这个问题的一个简单解决办法是对这些文件进行加密。有些操作系统提供了加密文件系统来防止这类问题。OS X 可以通过一个叫 FileVault[1]的功能来加密整个硬盘，类似的，Ubuntu 有 EncryptedHome[2]可以在你的 home 目录下挂载一个加密分区。Windows 也在文件系统[3]中提供了加密文件夹的机制。

有了加密的文件夹，可以将文件放到加密的目录下，然后在原始位置创建一个软链接，这样 AWS CLI 在加密目录被挂载之后就能读取这些文件。如果你的整个硬盘都被加密了，正如那个 FileVault 案例，那就不需要任何其他的步骤。

---

1 https://support.apple.com/en-us/HT204837

2 https://help.ubuntu.com/community/EncryptedHome

3 http://windows.microsoft.com/en-us/windows/encrypt-decrypt-folder-file#1TC=windows-7

设置多重身份认证

作为一个附加的安全层级，除提供密码或者其他证书以外，可以要求你的账户下的用户进行认证时提供一个生成的安全码。你可以使用很多兼容的应用和设备来生成这些码，它们是临时的而且每 60 秒更新一次。这通常叫作多重身份认证（Multifactor Authentication，MFA）。

移动设备可用的选择包括 Google 的 Authenticator 应用[4]（支持包括 AWS 的第三方账户）、Authy 应用[5]和 Amazon 自己的能在 Android 设备上运行的 Virtual MFA 应用[6]。另外，你可以从 Amazon 上花 12.99 美元购买一个硬件密钥表（physical key fob）[7]。

添加 MFA 对你的 AWS 账户来说是一个非常好的主意，而且你账户中的 IAM 用户也可以用它。对于那些拥有管理员权限但证书必须以明文形式保存（因为这样或者那样的原因）而且被频繁使用的用户而言，这可能是防止出现漏洞的一个好办法。

现在我们已经学习了关于账户安全的基础知识，下面就看一看无服应用可能会遭受的一些特定攻击。我们将深入理解这些攻击的工作原理，以及防范它们的方法。在本章的最后，你应该对这类无服单页应用可能会面对的攻击有相当的了解。

查询注入攻击

我们要研究的第一种攻击是查询注入攻击（query injection attack）。你可能听过这类攻击的一个更具体的类型：SQL 注入攻击。但这种特定的攻击形式可以多种多样，并不是所有的都是基于 SQL 的。

---

4 https://itunes.apple.com/us/app/google-authenticator/id388497605?mt=8

5 https://www.authy.com/

6 http://www.amazon.com/gp/product/B0061MU68M

7 http://onlinenoram.gemalto.com/

第 6 章中创建的自定义服务有一部分提出了 DynamoDB 扫描请求。在这个请求中，我们使用了 ExressionAttributeValues 参数将参数化的值添加到查询中。FilterExpression 字符串关联到这些值上。重新看这些代码，看看是否有些对你来说感觉很奇怪的东西：

learnjs/7000/services/lib/index.js

```
exports.dynamodb.scan({
  FilterExpression: "problemId = :problemId",
  ExpressionAttributeValues: {
   ":problemId": json.problemNumber },
  TableName: config.dynamoTableName
```

看完这段代码，你可能会好奇为什么不直接把 problemNumber 属性值添加到 FilterExpression 字符串中？为什么费力创建一个:problemId 表达式变量却只用一次？难道不是构建一个查询字符串更加简单？

与在浏览器中运行的 DynamoDB 请求不一样，我们的 Lambda 函数的 IAM 策略中没有任何东西限制对表中数据的访问。如果用户提交了一个带 problemId 的请求但它其实是表达式语句的一部分，这个请求将被添加到表达式的末尾。既然我们指望此表达式能限制某些记录的访问，它就可能会产生一个漏洞。比如，这个请求将会返回数据库中的所有数据：

```
$ curl -d '{"problemNumber":'1 OR problemId > 0'}' «endpoint_url»
```

下面这个请求将会返回所有属于 Cognito ID 为 abc123 的用户的记录：

```
$ curl -d '{"problemNumber":'1 OR userId = abc123'}' «endpoint_url»
```

正如你所见，人工构建数据库请求的查询语句可能会导致各种各样的问题，不管你使用的是否是 SQL 数据库。写一个使用变量名的查询，让 DynamoDB 替我们插入值，就可以阻止一次注入攻击。在请求中使用 ExpressionAttributeValues 字段，能确保提交像前面那样的查询，要么会产生一个错误，要么就不返回任何结果，因为没有任何记录的

problemId 属性是字符串"1 OR problemId > 0"。

接下来要讲的攻击也是类似的形式。不过，我们将会看到攻击者不是在服务中注入查询文本，而是在浏览器中注入 JavaScript。

跨站脚本攻击

进行跨站脚本（XSS）攻击需要添加<script>标签或者其他的 HTML 内容标记到 HTML 页面元素中。这会导致标记被执行，对于<script>标签，则意味着标签里面的 JavaScript 脚本将会被执行。因为单页应用使用大量动态 HTML，我们需要关注这种类型的攻击。

XSS 注入方法

2014 年，程序员杰米·汉金斯（Jamie Hankins）演示了很多提供 DNS 信息的网站都有的一个漏洞。DNS 记录是公开的，并且免费提供访问，所以有很多站点帮助人们轻松地在网上找到这类信息。不幸的是，一些站点是匆忙搭建起来的，它们就成为这起有趣而又富有启发性的恶作剧的受害者。

通过更新个人域名的 DNS 记录来包含 JavaScript 标签，汉金斯可以让许多站点跳哈林摇舞（Harlem Shake），同时还会播放音乐并且让页面元素随着鼓点移动[8]。虽然这个攻击只会影响查询站点 DNS 信息的用户，但是它暴露了一个明显具有潜在危险的 bug，很可笑。

我们的应用中有一些不可信的数据可能会成为这类攻击的目标。在理想状态下，我们的 DynamoDB 记录的 answer 属性应该是一个正确答案，但现在没有任何东西来保证它就是正确答案。不过应用逻辑会对此进行检查，但是这纯属为用户提供方便。任何用户只要他想这样做，都可以使用自己的 Cognito ID 写一条答案记录，而这个记录中并不包含正确

8 http://www.tomsguide.com/us/harlem-shake-hack,news-19595.html

答案，或者根本不包含 JavaScript。

正因为如此，我们必须小心处理这个数据。例如，当构建一个视图来显示它的时候，你会希望能确保通过使用 jQuery 的 `text()` 方法来设置元素用于显示的文本。这个方法在 DOM 上创建了一个文本节点，包含了 HTML 标签和其他特殊字符，以便它能够被安全地添加到页面中。使用其他方法，例如 `append` 或者 `prepend`，将可能会导致不安全的标记被添加上去，而产生一个 XSS 缺陷。

使用 web worker 沙盒化 JavaScript

现在，我们的应用遭受了一种自己施加的 XSS 攻击。因为用户的答案是在与应用同样的上下文中执行的，用户输入的代码可能会影响应用本身。虽然这不是一个安全问题，但它可能会带来一些十分奇怪的行为。看一看下面这个针对第 1 题（problem #1）的非常合理的答案：

```
learnjs = true;
return learnjs;
```

因为这段代码在定义 `learnjs` 变量时缺少 `var` 关键字，所以在我们的应用上下文中运行这段代码会重写应用的命名空间，导致如下的错误：

```
Uncaught TypeError: learnjs.buildCorrectFlash is not a function
```

为了防止用户代码影响我们的应用，可以在一个 web worker 中评估用户的答案。web worker 在主应用的一个独立线程中运行，我们通过发送和接收消息来与它通信。创建一个 web worker 来评估用户的答案，我们就可以把这段代码和应用的其他部分隔离。

为了创建一个 web worker，首先需要创建一个独立的 JavaScript 文件来规定这个 worker 的行为。这个 worker 文件需要赋一个函数给全局的 `onmessage` 变量，它在 worker 接收到一条消息时被调用。这个函数将由一个事件对象调用，该对象可以包含希望传给 worker 的任何数据。

worker 函数需要做的是判断答案的正误，返回一个响应。如果有异常发生，它可以直

接返回 false。在 public 目录下创建一个名为 worker.js 的新文件，并把这个函数添加进来：

learnjs/7100/public/worker.js

```
onmessage = function(e) {
  try {
    postMessage(eval(e.data));
  } catch(e) {
    postMessage(false);
  }
}
```

一旦创建 worker 文件，就可以把它集成到应用中，但是需要修改在第 3 章创建的 checkAnswer 函数的行为。checkAnswer 函数应该使用这个 web worker 来评判一个用户的答案。因为这个过程会是异步的，所以你可以返回一个 jQuery 的 Deferred，一旦判断出答案的正误，就让“Check Answer”按钮的点击处理函数更新用户界面。这个结果应该像下面这样：

learnjs/7100/public/app.js

```
function checkAnswer() {
  var def = $.Deferred();
  var test = problemData.code.replace('__', answer.val()) + '; problem();';
  var worker = new Worker('worker.js');
  worker.onmessage = function(e) {
    if (e.data) {
      def.resolve(e.data);
    } else {
      def.reject();
    }
  }
  worker.postMessage(test);
  return def;
}
```

这里我们在 worker 中使 postMessage 方法来发送用户的答案，该方法被包装在一个 JavaScript 函数内。注意，onmessage 处理函数是在 worker 中设置来接收这个响应的。

learnjs/7100/public/app.js

```
function checkAnswerClick() {
  checkAnswer().done(function() {
    var flashContent = learnjs.buildCorrectFlash(problemNumber);
    learnjs.flashElement(resultFlash, flashContent);
    learnjs.saveAnswer(problemNumber, answer.val());
  }).fail(function() {
    learnjs.flashElement(resultFlash, 'Incorrect!');
  });
  return false;
}
```

跨站脚本攻击有时候是攻击者预谋攻陷某个应用时采取的第一步。注入的脚本不仅可以读取或者修改浏览器的可访问资源，还能提交认证过的请求给 Web 服务，将认证证书保存为 cookie。还有一种技术称为跨站请求伪造（Cross-Site Request Forgery，XSRF），接下来将介绍这种攻击。

跨站请求伪造

跨站请求伪造需要使用保存在浏览器中的证书来发送认证过的请求给 web 服务。这类攻击经常和 XSS 攻击配合使用，用恶意 JavaScript 脚本模仿你，在你未知或者未经你同意的情况下执行一些操作。

这种攻击要求 Web 服务在浏览器上保存认证令牌，不管是保存在浏览器的 cookie 中还是保存在应用层。如何访问这些令牌取决于它们是如何存储的。基于 cookie 的令牌被简单地添加到任何匹配的对外请求中，因此攻击者需要做的就是制造一个有效请求，认证信息将由浏览器添加。如果是攻击应用层令牌，攻击者要么知道令牌存在哪儿，要么知道如何使用 JavaScript 来构建一个包含那些令牌的请求。

理解 cookie

当你向某站点发送请求时，站点的响应可以选择性地包含 Set-Cookie HTTP 头信

息。这个头的文本包含了一个值或者一个键值对，它可能包含例如过期时间、路径和域名选项的元数据。以下是一个例子：

```
Set-Cookie: sessionToken=sessionID_abc123; Expires=Fri, 13 Feb 2009 23:31:30 GMT
```

只要浏览器向之前返回带 Set-Cookie 头响应的站点回发一个请求，这些针对 cookie 的设置（比如 Expires）就会执行。所有仍然有效而且与请求的属性（比如域名和路径）值匹配的 cookie 都会被发送回服务器。所有之前收到的 cookie 都会作为一个名为 Cookie 的请求头的值被发送，如下所示：

```
Cookie: otherCookie=yes; sessionToken=sessionID_abc123
```

通过设置和接收 cookie，Web 服务就可以在用户的浏览器中保存该用户的状态。有时候，cookie 中的数据会被加密以防止用户或者第三方查看。当使用 cookie 来存储认证令牌时通常会发生这样的情况。

不用 JavaScript 实现 XSRF

虽然 JavaScript 看起来似乎是 XSRF 攻击的第一选择，但是有时候不用它也可能实现 XSRF 攻击。尽管不符合 HTTP 的约定，有些 Web 服务还是会提供 API，允许你执行一些有副作用的请求，这类请求中使用了 HTTP GET。在这些情形下，往页面中注入一个<img>或者其他资源标签足以发送这个请求。

不使用 JavaScript 创建一个 HTTP POST 请求也是可能的。通过注入一个隐藏的 HTML 表单到页面，你可以欺骗浏览器去发送一个包含该表单数据的 POST 请求。这就需要服务接受 POST 主体中编码后的表单数据。

在我们的应用中，任何注入到应用的恶意代码都可以通过和 AWS JavaScript SDK、应用本身，或者与两者交互，轻松地执行一个 XSRF 攻击。幸好，我们的应用中用到的 AWS API 使用了正确的 HTTP 语法，并要求任何修改数据的操作都必须通过 POST 发送。在构建自定义 Web 服务时，你也应该小心地遵守这些约定。创建一个响应 HTTP GET 请求修改

数据的服务，可能会给应用带来 XSRF 攻击的风险。

跨站请求和同源策略

一种可以防止简单 XSRF 攻击的方法是同源（same-origin）策略，所有现代浏览器都强制执行了此策略。这个策略限制了对 Web 上不同源的资源的访问。源是 URI scheme[9]、主机名和端口的组合。例如，以下所有的 URL 指向不同的源，即使它们可能都连接到同一个服务器：

1. http://example.com
2. http://www.example.com
3. https://www.example.com
4. http://www.example.com:8080

浏览器不仅使用同源策略来区分不同站点的 cookie，而且默认情况下它们还防止有副作用的 HTTP 请求（POST、PUT 和 DELETE）被发送到与当前载入页的不同源的站点。这意味着如果你用 URL http://learnjs.benrady.com 加载一个应用，并尝试发送一个 POST 请求到 http://www.google.com，浏览器将会拦截这个请求。

跨源资源共享

如果需要在不同源之间允许访问信息，可以使用跨源资源共享（cross-origin resource sharing，CORS）头来实现。响应此请求的 Web 服务需要在响应中加上 `Access-Control-Allow-Origin` 头来告诉浏览器允许这个请求。

你可能会好奇在响应中添加一个头怎么会起作用，毕竟在那个时候这个请求已经被接受并处理了。事实上，对于这个问题，浏览器会在发送实际请求之前，发送一个 preflight[10]

9 https://en.wikipedia.org/wiki/Uniform_Resource_Identifier#Syntax

10 https://en.wikipedia.org/wiki/Cross-origin_resource_sharing#Preflight_example

OPTIONS 请求，来查看 Web 服务会不会返回 CORS 头。如果当前源与头返回的值匹配，浏览器会发送原始请求到这个服务。如果不匹配，该请求会被拦截。

在第 4 章使用 Amazon Cognito 实现认证即服务中，当我们在 Google 开发者控制台创建一个应用时，必须指定被允许代表应用访问这个 Google API 的源。Google 要求我们指定这些源，实际上帮助了应用防止 XSRF 攻击。

现在我们已经知道应用在协议和应用层攻击上会受到怎样的攻击，让我们深入一些层级，看看网络传输安全对应用有何影响。尽管你可能不会经常用到这个层级的技术，但是构建安全的 Web 应用就必须理解如何通过互联网安全、私密地发送数据。

线路攻击和传输层安全

通过互联网发送数据时，不要指望它会被保密。发出去的每一个包就像一张小小的明信片，路上任何人想要看的都可以看。互联网这种天生的透明性让传输层安全（Transport Layer Security，TLS）变成最重要的 Web 安全技术之一。

HTTPS 是 HTTP 协议加上 TLS 或者 SSL。

与它的前辈，安全套接字层（Secure Sockets Layer，SSL）标准一样，TLS 提供了一种协商安全的加密通信通道的方式。它还允许通信的各方使用公钥加密验证他们的身份。每次在浏览器窗口上看到一个小锁样子的图标，告诉你请求正在被加密发送到正确地址，这都是 TLS 的功劳。因此，这一点很重要：考虑你的应用发送的所有 HTTP 请求，检查它们是不是在使用 TLS 来保障用户数据的安全。

Sidejacking 攻击

例如，在应用的某些部件中使用 HTTPS 可能会导致严重的问题。如果应用让用户通过

HTTPS 登录，但是其发布的身份令牌是通过纯文本方式传输的，就很容易招来 Sidejacking 攻击。这些攻击允许第三方模仿用户，并且趁认证令牌仍然有效的情况下冒充用户发送请求。这类攻击很难被检测到。

2010 年，一个针对 Facebook 的 Sidejacking 漏洞通过 Firefox 的浏览器插件被发布出来。它名叫 Firesheep，为一个由来已久的安全问题带来了非常需要的关注。虽然用户登录 Facebook 是受 HTTPS 保护的，和站点的交互却不是，所以任何人都可以从一个未受保护的请求中取走认证令牌，然后冒充那个用户，直到令牌过期。

2013 年，可能是因为 Firesheep 和其他漏洞引起了公众的广泛关注，Facebook 把 TLS 作为所有用户的默认访问方式。许多其他站点纷纷效仿，在很多场景中，HTTPS 现在已经是 Web 通信的默认方式。

高效使用 HTTPS

目前，我们的应用只在某些通信中使用 HTTPS。这有问题吗？为了找到答案，我们将彻查应用，找出哪一部分是安全的，而哪一部分不是。

我们的应用本身不是使用 HTTPS 加载的。访问应用时所用的 `http URL` 地址应该是第一个表征。这算不上是一个问题，因为应用本就应该是可公开访问的。它不包含敏感数据，用户也没有办法向托管在 S3 上的静态文件添加任何数据。

有可能 HTTPS 能为我们的用户提供一些私密空间，防止第三方看到他们在使用应用的哪一部分。在一些人权保护比较差的国家，这可能会是一个大问题。因为我们的整个应用是一次性加载，不是用户通过应用导览逐步加载的，这种流量中并没有太多的信息可以获取。唯一能确定的方式是，给应用加上 HTTPS 来对通信进行加密。

未加密的流量可能会暴露人们对公开站点的使用规律。

我们应用的登录流程使用 Google 的 OAuth2 端点[11]。在浏览器的“Network”标签下查看这个请求，可以看到这个端点是通过 HTTPS 提供服务的。类似的，我们发送来创建 Cognito 身份以及请求临时 AWS 证书的请求都是通过 HTTPS 提供服务的。

所有 Amazon 的 Web 服务 API 都是通过 TLS 保护的端点提供访问的。我们仅仅需要保证使用的是正确的端点。因为要使用 JavaScript SDK 来连接这些服务，所以需要用它来配置我们的应用如何与 AWS 连接。DynamoDB 默认使用安全连接，但可以将 `sslEnabled` 选项设置成 `false` 来禁止。你可以在设置区域（region）的那个参数对象中配置这个选项，但你应该不会想这么做。

使用 Wireshark

你可能想关闭加密的一个理由是希望能够在网络层看到应用的请求和响应。同样，要确切知晓你的应用是否是以纯文本的方式发送数据，一个好办法是自己检查。Wireshark[12]就是一种能为你逐个包地展现应用正在通过网络发送的内容的工具。它有先进的分析工具，能重组 TCP 会话，根据来源、目的地、协议或者成百上千的其他条件来筛查流量。

最后，当应用使用 AWS SDK 调用 Lambda 函数时，你可以看到这些请求是通过 HTTPS 发送的。这个请求到底使用哪个端点取决于 Lambda 服务所在的区域。你可以查看“Network”标签下对应的请求，看到该区域的具体端点。我们通过 API 网关访问 Lambda 服务所使用的公开 URL 地址也是受 HTTPS 保护的。

至此，我们有理由确信应用中的所有敏感数据在传输时都是受 HTTPS 保护的。我们已经验证了应用产生的所有网络请求都使用的是 HTTPS，应用加载是个例外，但是它不包含任何用户信息。接下来，我们来看一种能让你付出沉重代价的攻击。

---

11 https://accounts.google.com

12 https://www.wireshark.org/

拒绝服务攻击

拒绝服务（DoS）攻击是指攻击者用海量的恶意网络流量冲击一个传统 Web 应用，故意拖慢客户端或者对请求的响应。了解了应用的基本工作机制之后，攻击者可以为应用创建远超出其承受力的工作负载，让一个或多个应用层崩溃，并阻止任何用户使用这个应用。在 TCP 包层、HTTP 协议层或者应用请求层都可以实施这种攻击。不论在哪一层实施攻击，只要创建一个请求所需要做的工作和处理该请求所需要做的工作不对称，就能制造出一个可行且有效的 DoS 攻击。

由于应用是构建在 AWS 之上的，我们的问题有所不同。相对于让应用奔溃，更可能的情况是攻击者导致你的应用增加了很多计算开销，在月底收到 AWS 账单时候让人倒吸一口凉气。为了防止这种情况发生，我们必须好好研究应用的各个部分，找出保护它们的方法。

用 CloudFront 保护 S3

也许在面对 DoS 攻击时我们需要做的最基础的保障是让用户能加载应用。一旦应用被加载，我们就有很多可用的方案来减缓持续的攻击，但如果应用始终无法在用户的浏览器中加载，就会一切尽失。一种确保应用即使在被攻击时也能成功加载的方法，是使用 Amazon CloudFront 服务存储在 S3 中的应用静态内容。

除了作为内容传输服务的简单功能，Amazon 还将一些保护功能集成到 CloudFront 中，让它能检测并减缓 DoS 攻击（虽然 Amazon 不喜欢谈起它）[13]。既然我们的应用不过是一堆提供给浏览器的静态内容，我们就可以使用 CloudFront，利用这些保护功能来提供对应用的访问。

为了设置 CloudFront，创建一个新的 distribution 来控制内容是如何被缓存、内存来自哪里以及内容的访问是如何被记录的。这个过程很简单，并且在 AWS 文档中有详细的说明[14]。

13 https://forums.aws.amazon.com/message.jspa?messageID=253609

14 http://docs.aws.amazon.com/gettingstarted/latest/swh/getting-started-create-cfdist.html

选择“Restrict Bucket Access”（限制桶访问）选项，因为我们不希望攻击者能够绕过CloudFront直接访问S3存储桶。

一旦创建好distribution，就需要修改在第1章中创建的“CNAME DNS”项，将其指向CloudFront的端点。这个端点看起来会`d707s2ze1o682.cloudfront.net`这样。除非你临时允许直接访问S3，在DNS修改生效期间这个站点可能会不可用。一旦DNS修改生效，我们的应用就会通过CloudFront提供服务了。

创建CloudFront distribution，不仅将保护应用不受DoS攻击，而且将为用户提供更快的访问。全球有很多CloudFront的边缘位置（edge location）[15]，为你在该地区的用户提供本地访问。

分布式拒绝服务攻击

为了让攻击难以被挫败，互联网上的好事之徒通常同时从不同位置发起DoS攻击。来自一个IP地址的有害请求通常很容易检测，然后使用硬件或者软件防火墙屏蔽。从一组看似随机的IP地址发送过来的请求则很难处理。这类型的攻击被称为分布式拒绝服务攻击（Distributed Denial-of-Service，DDoS）。

可扩展服务和用户身份

要通过DoS攻击DynamoDB和Lambda比较难，但并不是不可能的。为了做这个事情，攻击者需要创建可用于发送请求的Cognito身份，要不然这些请求会被拒绝。

Cognito、DynamoDB和Lambda是相对较新的技术，所以很难讲哪种攻击是可能的。你可以想象一种可能的应用层攻击，它通过身份服务商（比如Google）创建一个临时用户，然后通过这个用户用伪造记录填满我们的DynamoDB表来实施攻击。

为了检测这种攻击，我们可以使用Amazon的CloudWatch服务，当发现不正常的使用

15 https://aws.amazon.com/cloudfront/details/

模式时就发送警报。比如，一个有关 DynamoDB 写入数量的 CloudWatch 警报可以帮助我们检测这类型攻击。我们将会在下一章深入介绍 CloudWatch。

假设我们能在攻击发生时检测到它，那么可以针对它做些什么呢？可以做的一件事是修改控制 Cognito 用户访问 DynamoDB 或 Lambda 的 IAM 策略的 `Condition` 语句。具体来说，剔除创建所有这些伪造数据的用户，可以防止攻击者写入新的记录。攻击者将不得不在重启攻击之前用身份认证服务商创建一个新身份。

重新部署

在本章中，我们了解了一些在搭建无服单页应用时会遇到的不同类型的攻击。虽然在安全领域并没有银弹，但对可能的威胁了解得越多，当它们发生时就处理起来心里就越有底。

如果你还没按照本章所述的这样做，那就赶紧部署本章前面添加的 web worker 修改吧。这样应该可以帮助防止用户的 JavaScript 答案和我们应用之间的非正常交互。

下一步

至此，你已经了解了一些针对无服 Web 应用的常见攻击方式，下面是一些你可能会感兴趣的延伸话题。我没有在本书展开讨论，你可能希望在继续学习下一章之前自行探索下这些话题。

使用已登录的 URL

使用 AWS SDK，你可以生成已登录的 URL 来提供对受限文件和资源的临时访问。你可以在 S3 上保存这些资源，然后使用 CloudFront 缓存它们。查看 AWS 文档可以了

解更多内容[16]。

战胜 DDoS

尽管本章谈到这个话题，但战胜 DDoS 攻击没有这么简单。要查找更全面的关于如何让运行在 AWS 上的应用战胜这类攻击的资料，请参阅 Amazon 白皮书“AWS Best Practices for DDoS Resilience”[17]。

下一章，我们将会看到应用流行之后会发生什么。我们将规划它在扩张到数百万用户时的运行成本，然后了解几个当用户增长时用于监控应用的工具。

16 http://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/private-content-signed-urls.html

17 https://d0.awsstatic.com/whitepapers/DDoS_White_Paper_June2015.pdf

第 8 章

扩容

我们的应用现在实现了基本功能：回答 JavaScript 编程题。你准备好发布它了吗？

比如，我们把它分享给朋友，朋友再分享给他们的朋友，一传十，十传百，最后数百万用户使用这个应用。它的性能会如何？支持这么多用户需要多少成本？应用每天执行上万次操作，我们该如何监控？

如果不是应用的发展速度超出你的想象，你可能都不会考虑这些问题。遇到这些问题是好事，但仍然需要解决它们。在本章中，你会发现一些解决这种问题的可行方案，并学习如何使用它们。这样，当你的应用发展到拥有大量用户时，就能马上采取行动解决这些问题。

监控 Web 服务

就像抚养自己的孩子一样，你希望看着应用成长、成功，偶尔的失败。不然，可能等到你知道发生了什么时，已经为时已晚！用于监控无服应用的应用程序和传统的应用程序在形式上是不同的，因为它没有应用服务器作为信息的中心访问点。这不是什么缺点，只

意味着需要采取不同的手段。

首先，我们需要监控正在使用的 Web 服务。可以使用另一个 Web 服务来做这件事。Amazon 的 CloudWatch 服务让我们可以基于 AWS 和任何第三方服务的统计数据进行监控并发送告警消息。

CloudWatch 是一个 Web 服务，它根据 AWS 和用户自定义标准进行监控和发送通知。CloudWatch 原生支持很多 AWS 服务，只需要在 CloudWatch 控制台中单击几个按钮或运行一条 AWS CLI 命令就能监控这些服务。它还提供一个 API，你可以用来发送自己的统计和日志消息。不管你是否关心内置或自定义指标，只要你的统计数据中有一条超过阈值，就可以发送告警邮件到指定的邮箱。

监控容量限制

还记得第 5 章讲的预设吞吐量吗？我们当时设定了 DynamoDB 表的读写限制。设置了这些限制后你可能会想到的第一个问题是“我怎么知道自己是不是超过了这个限制呢？”如果没有监控这个服务，你只能从用户愤怒的抱怨中知道。

幸好，我们可以创建一个 CloudWatch 警报，如果超出了读写容量限制而导致对 DynamoDB 的请求失败时，我们会得到通知。但是在创建警报之前，需要为警报创建一个发送的去处。对于 CloudWatch，你有两个选项：将警报发送到 Amazon 的简单通知服务（Simple Notification Service，SNS），或者是 Amazon 自动扩容策略。我们只是想在这些情况发生的时候得到通知而不希望自动改变任何设置，所以将这个警报发送到一个 SNS 主题（topic）。

你可能想知道这个神秘的 SNS 到底是什么。我们真的必须引入另一个服务吗？为什么不发送一个文本或者邮件？使用 SNS，两者你都可以做到，甚至更多。SNS 也称作消息推送服务，它允许你创建若干个主题来接收通知和推送通知给多个订阅者。订阅者可以通过多种协议处理这些通知，例如邮件、文本消息、或者一个 HTTP POST（有时被称作网络钩子）。

你甚至可以把这些通知发送给 Amazon 的简单队列服务（Simple Queue Service，SQS）[1]，或者如果你想用编程的方式消费它们，也可以将通知发送到一个 Lambda 微服务。

虽然这些配置项可能有点多，在如何处理发送到 SNS 的告警上，Amazon 提供了很大的灵活性。但是这意味着创建警报的第一步是新建 SNS 主题。我们可以通过 AWS CLI 来完成，就像使用其他服务一样。这里所用的命令是 `sns create-topic`，像这样使用：

```
$ aws --profile admin sns create-topic --name EmailAlerts
```

sspa 脚本与 CLI

在前面的章节中，我们用 sspa 脚本与 AWS 交互来创建 DynamoDB 表、IAM 策略和其他资源。在背后，sspa 脚本其实是使用 AWS CLI 来执行这些操作的。

本章将展示如何更直接地使用 AWS CLI，而不是使用 sspa 脚本和 AWS 交互。我们在本章做的大部分事情都很简单，所以这是介绍 CLI 的好机会。当你对它的使用已经很适应时，可以返回查看 sspa 脚本，看看它在本书的前几章帮你做了什么。

这条命令返回 ARN 主题，我们在接下来的几步会用到。一旦创建这个主题，就可以添加一个邮件订阅。使用 sns subscribe 命令并提供一个协议和邮箱地址，像这样：

```
$ aws --profile admin sns subscribe \
    --topic-arn «your_topic_arn» \
    --protocol email  \
    --notification-endpoint you@example.com
```

执行这条命令后，立刻会有一封邮件发送到你指定的邮箱。这个邮件包含一个确认链接，你需要点击它来完成订阅。完成之后，就可以发送警报到这个主题了。为了配置新的 CloudWatch 警报，你可以用 AWS CLI 中的 `cloudwatch put-metric-alarm` 命令。这里有一个例子：

```
$ aws --profile admin cloudwatch put-metric-alarm \
    --actions-enabled \
```

1 https://aws.amazon.com/sqs/

```
--alarm-name 'DynamoDB Capacity' \
--comparison-operator GreaterThanOrEqualToThreshold \
--evaluation-periods 1 \
--metric-name ThrottledRequests \
--namespace AWS/DynamoDB \
--period 60 \
--statistic Sum \
--threshold 1 \ --alarm-action «your_topic_arn»
```

这个警报监听一个名为 ThiottledRequests 的 DynamoDB 指标，统计因为应用的容量不足而失败的 DynamoDB 请求数。这个命令中的其他参数指定该警报抽取 60 秒时间段的数据样本，并计算这个时间段中所有数据点的和。如果失败的请求数超过或等于阈值，警报会被触发，导致一个通知被发送到我们刚才创建的 SNS 主题，该主题是在 alarm-action 选项中设置的。如果你收到一个警告，就可能会查看应用的查询使用模式，并因此相应地提高容量限制。

当出现问题时我们能收到通知对其解决是有帮助的，但是如果在出现苗头之前就收到通知会更好。幸好，我们还可以监控另一个 DynamoDB 指标。例如，通过查看 ConsumedReadCapacityUnits 和 ConsumedWriteCapacityUnits 指标，可以在应用超过容量限额的某个比例（例如 80%）时触发告警。这可以帮助我们在开始发生数据库操作失败之前发现容量问题。

Amazon 限制在每个 UTC 日历天中最多能对容量上限做 4 次调整。

有一点需要注意，告警是全局的，也就是说在任何一张 DynamoDB 表中只要这个指标超过阈值就会触发警报。为了仅对 LearnJS 的生产环境用表产生告警，我们需要通过 cloudwatch put-metric-alarm 命令的 dimensions 参数添加一个维度。维度允许你限定只对某些资源或操作产生告警，并且大多数 DynamoDB 指标允许你在创建警报的时候设置一个 TableName 维度。可以查看 DynamoDB 指标文档了解详情[2]。

2 http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/MonitoringDynamoDB.html#dynamodbmetrics

创建付款警告

除了监控应用的网络服务性能，你还可以监控你的信用卡。通过在 CloudWatch 里创建付款警报，每小时都可以检查你的 AWS 预估收费是否超过额度。有了这个警告功能，就能在月底收到巨额账单之前采取一些措施。下面的例子展示了如何创建这样的警告：

```
$ aws --profile admin cloudwatch put-metric-alarm \
    --alarm-name 'Billing Alarm' \
    --comparison-operator GreaterThanOrEqualToThreshold \
    --dimensions "Name=Currency,Value=USD" \
    --evaluation-periods 1 \
    --metric-name EstimatedCharges \
    --namespace AWS/Billing \
    --period 3600 \
    --statistic Maximum \
    --threshold 100 \
    --actions-enabled \
    --alarm-actions «your_topic_arn»
```

我们在这个警告里设置的阈值是 100 美元。它可以复用我们在其他警报里使用的邮件通知 SNS 主题。和 AWS/DynamoDB 命名空间不同的是，`EstimatedCharges`（当前的）是你能在 AWS/Billing 命名空间里监控的唯一指标。但是，要注意查看更新文档[3]。

意外的开销

如果有一天早晨醒来，发现你的付款警告没有起作用，一夜之间生成巨额 AWS 账单，千万不要惊慌。在关闭或者关停所有疯涨的服务之后，联系 Amazon 寻求帮助[4]。根据具体情况，Amazon 或许会减少或者免除你所有意料之外的费用，并和你一起防止将来再次发生这种错误。

Amazon 希望客户们满意，所以如果你遇到麻烦，他们很乐意提供帮助。AWS 并不是一场骗局，趁着你犯错、因为别人的错误受牵连或者遭受网络攻击的时候，骗走你一

3 http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/billing-metricscollected.html

4 https://aws.amazon.com/contact-us/

大笔钱。Amazon 希望你持续地使用它的服务，并且让你觉得物有所值。

除这两个例子之外，还可以为 AWS 很多不同的服务和指标设置警报。你可以创建新的 SNS 主题来用更直接的方式发送提醒，例如文本消息。你也可以创建自定义服务来程序化地使用通知，在超出阈值时自动采取措施。AWS 文档[5]提供了指标和命名空间的完整列表。

分析 S3 的流量

我们已经有很好的办法来监控所使用的网络服务，但还是需要有一个途径来获知应用本身在做些什么。在传统的 Web 应用里，我们通常会在应用服务器前设置一个专门的 Web 服务器，如 Apache 或者 Nginx。除一些任务之外，这个 Web 服务器还能提供对监控流量的日志的访问。

当然，用传统的方式，你也需要一种方法能在应用水平扩展合并 Web 服务器上的所有日志。因为我们用 Amazon S3 来支撑应用，不存在这个问题。然而，S3 产生的日志和传统 Web 服务器产生的有所不同。所以我们来看看它们，学习一些提取有用数据的常见技术。

记录 S3 请求

正如第 1 章开篇所述，你可以用一个 Amazon S3 存储桶（bucket）作为一个静态 Web 主机，我们的应用就是这样做的。Amazon 在 S3 中提供一个装置将所有请求记录到这个存储桶中。为了实现这一点，你需要创建另一个 S3 存储桶，然后配置这个 Web 存储桶向之前的那个桶写日志。然后你可以从这个 S3 存储桶下载日志，或者构建其他分析工具直接访问日志。

为了配置这个桶，要用 AWS CLI 中的 `s3api website` 命令。首先，我们需要创建一个存储桶来放日志，并授予 S3 向这个桶写日志的权限。我们之前使用的 `sspa create_bucket` 命令将配置这个桶作为一个 Web 主机，所以你可以用 AWS CLI 的 `s3 mb`

5 http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/CW_Support_For_AWS.html

命令，像下面这样：

```
$ aws --profile admin s3 mb s3://learnjs-logging.benrady.com
make_bucket: s3://learnjs-logging.benrady.com
```

注意，因为你随后没有使用 s3 website 命令，所以这个桶不会通过 HTTP 提供它的内容（这是好事）。接下来，我们需要授予写桶的权限。Amazon S3 会将日志文件写到这个桶，使其作为预定义的日志分发 Amazon S3 组的成员。为了能这样操作，你必须通过 AWS CLI 的 s3api put-bucket-acl 命令在这个桶里开启合适的访问控制限制。注意，这是一个与创建容器的命令不同的顶级命令（s3api）。你可以看一下完整的命令：

```
$ aws s3api put-bucket-acl \
   --bucket learnjs-logging.benrady.com \
   --grant-write URI=http://acs.amazonaws.com/groups/s3/LogDelivery \
   --grant-read-acp URI=http://acs.amazonaws.com/groups/s3/LogDelivery
```

既然存储桶已经可以写入日志，就需要配置激活了 Web 服务的 S3 存储桶来使用这个新的桶来记录请求。所以，现在是时候使用 s3api put-bucket-logging 命令了。

S3 组（如日志分发）和 IAM 组不是一回事。

我们准备用-cli-input-json 参数，并通过一个外部.json 文件控制这个配置，就不用在命令行里传入所有参数了。这个方法与第 7 章中使用 sspa 脚本的方法一样，所以我们还是在 conf 目录中存放配置文件。

你可以通过--generate-cli-skeleton 参数为这个命令创建一个简单的配置文件。运行这些命令，在 conf/s3 目录里创建配置文件：

```
$ aws s3api put-bucket-logging \ --generate-cli-skeleton >
conf/s3/«your.bucket.name»/logging.json
```

打开生成的文件看一看，填上（或者删除）这些值：

learnjs/8000/conf/s3/learnjs.benrady.com/logging.json

```
{
   "Bucket": "",
```

```
    "BucketLoggingStatus": {
        "LoggingEnabled": {
            "TargetBucket": "",
            "TargetGrants": [
                {
                    "Grantee": {
                        "DisplayName": "",
                        "EmailAddress": "",
                        "ID": "",
                        "Type": "",
                        "URI": ""
                    },
                    "Permission": ""
                }
            ],
            "TargetPrefix": ""
        }
    },
    "ContentMD5": ""
}
```

顶层字段是 put-bucket-logging[6]命令的参数。先用 S3 Web 存储桶的名字填写 Bucket 字段。BucketLoggingStatus 对象控制日志被写到何处以及可以被谁访问。ContentMD5 字段只不过是该请求的一个签名值，可以删除。

在 LoggingEnabled 对象里面，有三个字段。TargetBucket 是日志桶的名字，是你应该填入的。TargetPrefix 允许你在目标桶里指定一个子目录来写日志。如果你想在不同的应用中复用一个日志桶，这个字段非常有用。

TargetGrants 字段可以用于给某个用户或组授权。不过在我们的例子中，我们想要这个 TargetGrant 置空。这是因为 S3 是一个 Amazon 最悠久的服务之一，它不能像其他服务一样和 IAM 服务集成。除了通过 IAM 授权访问，S3 有它自己的安全模型，但是我们不需要使用。全部改完之后，你应该得到一个这样的配置文件：

6 http://docs.aws.amazon.com/cli/latest/reference/s3api/put-bucket-logging.html

learnjs/8010/conf/s3/learnjs.benrady.com/logging.json

```
{
   "Bucket": "learnjs.benrady.com",
   "BucketLoggingStatus": {
      "LoggingEnabled": {
         "TargetBucket": "learnjs-logging.benrady.com",
         "TargetGrants": [],
         "TargetPrefix": ""
      }
   }
}
```

现在执行这个命令来为我们的 Web 存储桶配置日志。

```
$ aws --profile admin s3api put-bucket-logging \
--cli-input-json "file://conf/s3/learnjs.benrady.com/logging.json"
```

S3 访问日志是按批分发的。一个请求的日志需要好几个小时才能分发给 bucket 桶。Amazon 不保证这些日志是完整的[7]，只是称它为一个“尽力而为”系统。既然我们要聚合这些日志里的数据，这就不是问题。

分析 S3 日志

既然我们从 S3 中获取到日志，那么如何处理它们呢？随着应用的发展，我们需要强大的工具来分析它所处理的数百万请求。日志中的数据太多，根本没法通过阅读日志来知道发生了什么。我们必须找办法聚合这些数据来看到事情的全貌。

幸运的是，不需要昂贵的专业日志分析系统，用你面前的这些工具就能分析 AWS S3 访问日志。使用和我们部署站点时相同的 `s3 sync` 命令，可以快速下载收集到的应用日志文件。下载它们之后，我们可以用一些简单（也很强大）的命令行工具来分析。只要使用得当，这些工具能在瞬间轻松消化上 GB 的日志数据。为了从你的日志桶下载日志到 `logs` 目录，执行下面这些命令：

```
$ mkdir logs
```

7 https://docs.aws.amazon.com/AmazonS3/latest/dev/ServerLogs.html#LogDeliveryBestEffort

```
$ aws s3 sync s3://learnjs-logging.benrady.com/ logs
```

因为S3日志是按批分发的，单个日志文件永远不会被更新。这很棒，因为s3 sync命令只会下载没有保存在本地的文件。所以随着应用程序的流量越来越多，日志文件会变得越来越大，但是你不需要为了获取最新日志一遍又一遍下载它们。用*nix命令行工具和列式日志文件，可以快速从巨大数据集中提取有用的信息。你可能早已习惯通过这些工具做这种数据分析，让我们快速看一看在S3给我们的日志格式上这些工具有哪些用法。

如果打开其中一个日志文件，你会发现每一行都很长，而且有很多列，每一列代表请求的不同属性。下面有一个请求的日志文件示例，一行中的每一列用空格隔开。

```
1c6655153443675143ce57960dd49971b0745bdd3175be16afd5229a33ded86f
learnjs.benrady.com
[13/Jan/2016:04:30:48
+0000]
98.223.150.212
1c6655153443675143ce57960dd49971b0745bdd3175be16afd5229a33ded86f
42AFE21FCF9C87AA
REST.GET.LOGGING_STATUS
-
"GET
/learnjs.benrady.com?logging
HTTP/1.1"
200
-
527
-
37
-
"-"
"S3Console/0.4"
-
```

这样查看日志时你会注意到的第一件事就是，一些数据字段散布在多个列里。例如，第3个字段是请求时间，它占用了两列，因为在时间和时区偏移之间有一个空格。如果我们将这条日志信息拆分为字段而不是列，并进行标注，就更容易发现发生了什么。

```
Bucket Owner: 1c6655153443675143ce57960dd49971b0745bdd3175be16afd5229a33ded86f
```

```
Bucket: learnjs.benrady.com
Time: [13/Jan/2016:04:30:48 +0000]
Remote IP: 98.223.150.212
Requester: 1c6655153443675143ce57960dd49971b0745bdd3175be16afd5229a33ded86f
Request ID: 42AFE21FCF9C87AA
Operation: REST.GET.LOGGING_STATUS
Key: -
Request URI: "GET /learnjs.benrady.com?logging HTTP/1.1"
HTTP Status: 200
Error Code: -
Bytes Sent: 527
Object Size: -
Total Time: 37
Turn Around Time: -
Referrer: "-"
User-Agent: "S3Console/0.4"
Version ID: -
```

S3 日志中的“-”代表数据对于那个请求不可用。

大部分*nix 命令行工具都能轻松处理行式和列式存储的文本。使用像 uniq 和 cut 这样的工具，可以快速高效地将这种数据切分为你想要的任何形式。我们用几个例子来理解如何切分数据，并且看看能从这些日志文件里得出哪些启示。

Bash 管道

Bash shell（或者其他 shell，比如 zsh）可以轻松使用“|”操作符（也称为管道操作符）将命令字符串连接在一起。管道操作符能将一个程序的输出作为另一个程序的输入。你可以把许多不同的程序组合在一起，轻松创建出进行变换、分析、转化、查询和存储数据的工具。这些程序通过管道绑在一起，这样的组合经常被称为管道（pipeline）。

响应代码频率

为了分析这些日志，我们准备创建一系列 Bash 管道。我们将用一些命令行工具来筛选想要的数据，对它们排序和分组以便分析。

所有这些管道都以一条打印日志文件内容的命令作为开始。通常，我们可以用 cat 命令，它将所选择的文件串起来，并打印出来供剩下的管道使用。如果我们想要分析所有文件，使用“*”就能将它们全部选中，如下所示：

```
$ cat logs/* | cut -d ' ' -f 13 | sort | uniq -c
     94 200
      8 304
     20 404
```

管道中的下一个命令是 cut，它能方便地获取单列或者多列内容。通常，cut 用制表符作为定界符。参数-d 允许我们指定空格符作为定界符。参数-f 指定选择哪一行。将它设置为 13，表示从日志消息中挑出 HTTP 状态码，并且一次一行把它们打印出来。

最后我们要做的是统计各个状态码出现的次数。要达到这个目的，我们可以用带有-c 参数的 uniq 命令，但是 uniq 要求它的输入是经过排序的，所以我们用 sort 来排序。

这条命令行输出的信息告诉了我们需要知道的一切。如果你想看漂亮的图表，这些工具可能不适合你。但是这个输出告诉我们有 94 个 HTTP 200 响应，20 个 404 响应和 8 个 304 响应。非常有趣。想知道 304 是什么吗[8]？哦，对了！304 这个 HTTP 状态码帮你在数据传输上省钱。真是太好了，我们有一些 304。

热门资源

AWS 分批发送这些日志，每个文件都有一个日期时间戳。用不同的通配符选择文件，就可以根据时间或者日期筛选结果。例如，用方括号匹配器“[]”可以选择特定的文件子集，展示哪些资源在 1 月 11 日、1 月 12 日和 1 月 13 日被请求得最多：

```
$ cat logs/2016-01-1[123]* | cut -d ' ' -f 11 | sort | uniq -c| sort -rn
     12 /?location
     11 /vendor.js
     11 /app.js
     10 /images/HeroImage.jpg
```

8 https://httpstatuses.com/304

```
    9 /favicon.ico
    9/
    8 /learnjs.benrady.com?logging
    6 /learnjs.benrady.com?versioning
    4 /learnjs.benrady.com?website
    4 /learnjs.benrady.com?tagging
    4 /learnjs.benrady.com?requestPayment
    4 /learnjs.benrady.com?prefix=&max-keys=100&marker=&del...
    4 /learnjs.benrady.com?policy
    4 /learnjs.benrady.com?location
    4 /learnjs.benrady.com?lifecycle
    4 /learnjs.benrady.com?cors
    4 /learnjs.benrady.com?acl
    4 /?replication
    4 /?notification
    1 /worker.js
    1 /index.html
```

在这例子中，你会发现请求的一些资源不在我们应用里。它会是网络爬虫吗？或者是更可怕的东西？如果用一些简单的命令能快速从日志中拉取数据，很容易发现这些现象。

每日用量

了解用户在什么时候访问你的应用通常是有帮助的。如果你一天能接纳 10,000 个新用户，但是他们都倾向于在晚上 5 点到 8 点之间出现，你对资源采取的分配方式和他们在 24 小时内均匀出现的情况肯定要有所区别。这个信息就在日志里，只需要你自己取出来。

要理解管道的作用，采用一次运行一个命令，然后每完成一步再在后面插入新命令，直到能看到最后输出这种方式会很有帮助。如果你想尝试一下，可以用下面的管道（太长了，一行放不下）：

```
$ cat logs/* | awk '{ print $3 }' | tr ':' ' ' | \
    awk '{ print $2 ":" $3}' | sort | uniq -c
  16 04:25
   1 04:27
  30 04:30
  17 04:31
   3 04:32
```

```
25 04:37
20 04:38
 2 04:46
 8 15:24
```

这里我们用到另一个命令，awk。Awk 是一门非常成熟的编程语言，不过它多被用于*nix 环境的文本处理。精炼的语句使它成为 Bash 管道中文本处理的极佳选择。在这个例子中，有两处用到 awk。先是调用 awk 选出请求时间戳中的日期和时间部分。时区不在这一列，但对于我们要做的事情并未造成什么影响。

选出日期和时间之后，通过 tr 命令来用空格替代冒号。也可以用 sed，但是对于这种简单的替换操作而言，tr 的语法更短。我们把时间和日期切分成列之后，接下来把它传递给第二个 awk，提取小时和分钟信息，放到一个字符串中。然后对它们排序，并使用 uniq -c 按照一天中的“小时:分钟”这种形式对筛选出的日志中的所有数据分组。

从 S3 日志中获取的日志文件可能看起来平淡无奇，但是如果使用一些简单的命令行工具，它们就会告诉你一些奇妙的事情，比如用户是否使用你的应用，什么时候使用以及如何使用。如果想不断改良你的应用，就必须倚赖告警和像这样的收集数据的工作。不然，你就没办法知道所做的改动是否达到了想要的效果。既然能看到用户如何使用我们的应用，是时候优化它了，我们将在下一节中做这件事情。

优化应用，实现增长

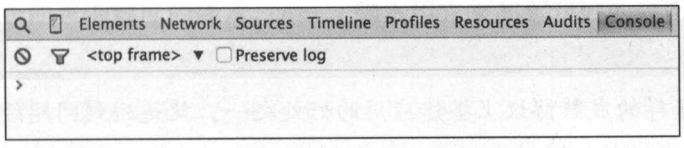
由于在应用中使用了高可伸缩的服务，我们可以将其交付给上百万的用户，最不济多花点钱也能办到这一点。但是通过优化我们的单页应用和它使用的服务，可以让花出去每一分钱都能起到更大的作用。另外，给 Amazon 再多的钱，应用的某些特征也得不到改善，比如如果要改变像应用加载时间这样的特征，需要我们自身理解应用以及采取行动。

通过缓存降低成本和加载时间

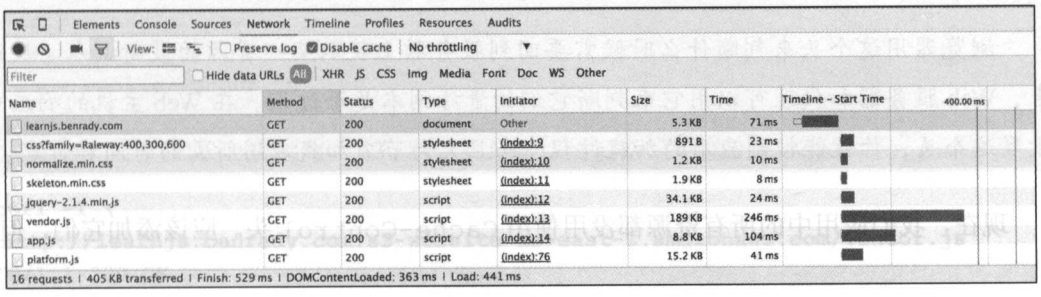
在 Web 分析的世界里，流失率（bounce rate）描述了访问过页面或者应用却没有产生任何交互行为的用户所占的百分比。好比一位顾客走进一家商店，但是什么也没看上，逛一圈就出去了。长时间等待页面加载是导致用户离开站点的原因之一。

如果我们在开发一个单页应用，相对于传统 Web 应用它有一个很大的优点，就是只需要加载一次，但还是要在用户使用之前先加载。为了确保不让用户等待，我们需要保证应用在几秒内加载完成。让应用加载时间尽可能短，占用的空间尽可能少，将使应用被尽可能多的用户访问到。

开始优化应用加载流程之前，需要理解应用在做的事情。我们可以用浏览器内置的开发者工具。下面的例子使用的是 Google 的 Chrome 浏览器，但是你可以在其他浏览器（例如 Firefox 和 IE）上找到类似的功能。打开 Chrome 开发者工具时，会看到如下图所示的工具条。



单击“Network”标签，查看应用的生产环境 URL（不是 localhost）。在这个面板里，你可以看到看到浏览器代表我们的应用发送的所有请求，见下图。



从这里我们能看到应用发送了超过 12 个请求，数据量达 405 KB。可能比你预想的多，

是吗？幸运的是，我们可以在S3存储桶中添加元数据信息，给这些请求添加Cache-Control头，让随后的请求都从缓存里获取，而不是经由互联网从S3存储桶或者CloudFront的边缘位置中获取。

Cache-Control请求头在S3里通过元数据应用到桶的各个实例，默认情况下是不添加这些头的。在命令中curl使用参数-I，可以发送一个HEAD请求来看看哪些头被添加到特定的资源上了：

```
$ curl -I http://«S3_Bucket_Endpoint»/vendor.js
HTTP/1.1 200 OK
x-amz-id-2: «id_tag»
x-amz-request-id: B3FCCF13142A9CCD
Date: Wed, 20 Jan 2016 18:43:30 GMT
Last-Modified: Tue, 05 Jan 2016 02:36:22 GMT
ETag: "210f8bac1f6ada8d5ecf9ca74e176a6b"
Content-Type: application/javascript
Content-Length: 193464
Server: AmazonS3
```

缓存控制头

在HTTP这样的成熟协议上搭建应用的好处之一，就是碰到问题经常都有现成的解决办法。如果有关计算机科学中的难题的各种陈词滥调是对的，那么Web缓存失效一定是你真的不想自己处理的难题之一。幸运的是，这个问题经常能通过缓存控制头（Cache-Control Header）[9]来解决。

浏览器用这个头来判断什么时候需要回到服务器上获取某一资源的最新副本。反过来，Web服务器和代理可以用它来判断它们的资源副本是否过期。在Web主机的响应中设置这个头，你就能控制应用的加载过程，而且大大节省加载应用所需的时间和带宽。

现在，我们应用中的所有资源都没用使用Cache-Control头。应该添加它们，但是

9 https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching?hl=en

要格外小心。我们有两种资源，它们各自有不同的缓存策略。对于一些鲜有变化的资源，例如 vendor.js 文件和英雄图片，我们希望强缓存。一旦用户下载了这些资源，就不大需要下载第二遍。告诉浏览器缓存这些资源，能省下一笔在 S3 传输数据的开销，并且使应用的加载更快。

对于一些频繁更新的资源，例如 index.html 和 app.js，我们希望每次加载应用时浏览器都检测新版本。这样的话，S3 在资源没有改变的情况下就返回一个 HTTP 304 响应（正如前面看到的），反之则发送新版资源。缓存这些频繁更改的资源，如果一个资源的缓存更新过了，而另一个有没有，可能会导致应用意外崩溃。例如，对 JavaScript 代码更新时，如果 JavaScript 引用了一个缓存的 HTML 中不存在的模板，将导致应用崩溃。

防止这些资源被缓存，也会让 S3 日志更有用。这意味着无论何时用户加载我们的应用，这些资源都将出现在日志里。对这些资源请求的次数将是衡量应用活跃度的可靠指标。

要为 Cache-Control 头添加元数据，可以使用 AWS CLI 命令 s3api put-object。这是 s3 sync 命令的更细粒度的版本，sspa 脚本用它来部署我们的应用程序。你需要在 Cache-Control 头中指定一个 max-age 值，以设置资源应缓存的时间长度（以秒为单位）。可以对要缓存的每个资源运行此命令，如下例所示：

```
learnjs $ aws s3api put-object \
  --profile admin \
  --acl 'public-read' \
  --bucket learnjs.benrady.com \
  --cache-control 'max-age=31536000' \
  --key vendor.js \
  --body public/vendor.js
```

现在，当获取这个资源时，S3 就包含了带有我们指定的值的 Cache-Control 头。

```
$ curl -I \
   http://learnjs.benrady.com.s3-website-us-east-1.amazonaws.com/vendor.js
HTTP/1.1 200 OK
x-amz-id-2: «id_tag»
x-amz-request-id: 7EB6F7458CC636C9
Date: Wed, 20 Jan 2016 18:44:08 GMT
```

```
➤ Cache-Control: max-age=31536000
  Last-Modified: Wed, 20 Jan 2016 18:43:47 GMT
  ETag: "210f8bac1f6ada8d5ecf9ca74e176a6b"
  Content-Type: binary/octet-stream
  Content-Length: 193464
  Server: AmazonS3
```

现在，此资源将缓存 31,536,000 秒（也即 365 天）。即使使用 `sspa` 脚本再次部署应用程序，此元数据仍将保留。

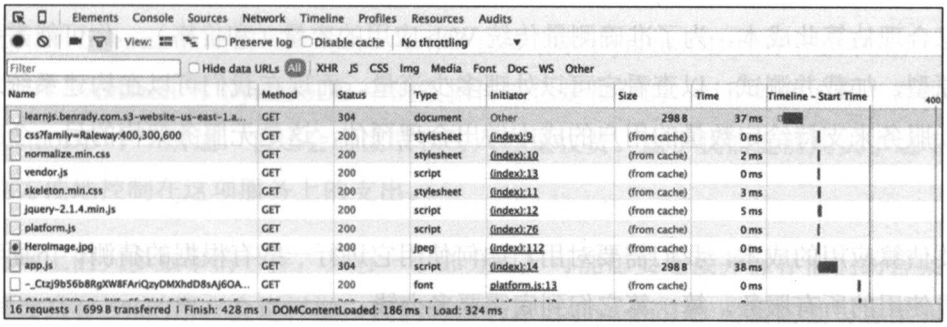
通过带版本号的文件名清除缓存

你可能想知道如果更改缓存了一年的文件会发生什么。更改像 `vendor.js` 这样的文件而不强制缓存更新，可能会导致应用发生错误。解决这个问题的一个简单的方法是停止使用这个缓存的资源——其实就是换一个名字！

凡是做过更改的资源，在其末尾附加版本号、标签或摘要，就可以轻松地更新缓存了较长时间的资源。在 `index.html` 中引用这些新版本资源，意味着当浏览器访问它们时，不会有该资源的缓存版本，因为资源名称将略有不同。然后它将一路返回到我们的 S3 存储桶获取最新版本。

重命名这些资源时，记得在 `index.html` 和 `tests/index.html` 文件中更新对它的引用。否则，你的测试将失败。

应用这些技术，应该就能缓存应用程序中的大部分资源了。一旦将适当的元数据应用于 S3 存储桶中的实例后，请关闭浏览器开发控制台中的“禁用缓存”按钮。加载应用程序两次（在地址栏中按 Enter 键），就可以看到典型用户在第二次加载应用程序时会做出什么样的请求。它应该是像下图所示的这样。



启用缓存后，我们的应用只检查 `index.html` 和 `app.js` 文件，从缓存加载所有剩余的资源。这大大减少了加载时间（本例从 441 ms 缩减到 324 ms）和数据传输速率，为用户创造了更好的体验，并降低了我们的成本。

当你按下 Ctrl-R 组合键或者 F5 功能键时，大多数浏览器都会无视缓存头。

云的成本

如果用户数量不多，无服应用的成本通常会下降到零。Amazon 提供的大部分服务是免费的[10]，只要你使用的资源保持在一定的范围之内。超出免费的范围所带来的增量成本通常极低。即使对于中等规模的应用，超过这些免费额度的成本每月也不过几美分。对于不管你只有一个用户还是 10,000 个用户，每月都收取固定费用的虚拟服务器主机解决方案而言，选用 Amazon 的服务可能更划算。

然而，虽然云服务的初始成本可能非常低，但它们经常随着系统的用户量而增加。如果你不留神，这些成本可能会快速累加。所有这些自动缩放的魔法尽在你的指尖，你怎么知道万一应用程序突然受欢迎，在月底不会收到一笔巨额账单？

避免此问题的第一步是了解每个用户的实际成本。可以根据 Amazon 提供的定价和容

10 http://aws.amazon.com/free/

量信息合理估算此成本。为了准确测量传统 Web 应用的容量（和价格），你可能需要构建一个原型，加载并测试，以查看它可以处理多少流量。而现在我们可以在构建系统之前对使用云服务来支持给定数量的用户的成本做出合理预估，这是无服务架构的另一个美妙的好处。

要估算应用的成本，我们需要对用户如何使用它进行一些有根据的猜测。下面我们就来看看使用的所有服务，算一算它们到底需要多少钱。

加载成本

当我们的应用程序第一次加载时，是由 S3 提供服务的。 S3 的作用只是一个网络服务器，应用启动之后就不起什么作用了。S3 存储这些数据、响应请求并传输数据，这些都是要收费的。如果你回顾刚才做的性能分析，会看到我们的应用第一次加载时向 S3 发出了 4 个请求，总大小为 261.5 KB。

由于我们添加了缓存，后续加载只花了几百个字节和几个请求。因此，当我们添加新用户时，通常只需要支付他们第一次加载应用的费用。可以根据 Amazon 提供的定价信息计算这些初始加载成本。以下是截至 2016 年 1 月美国东部 1 区 S3 服务的价格。

存储	请求（GET）	传输
0.03 美元/GB	0.004 美元/10K 请求	0.09 美元/GB

这意味着我们的应用的加载成本，包括请求和传输费用，每 1 万用户收取 0.24 美元。后面在 S3 中验证缓存的资源的请求，其成本还不到这个值的 1/100。在 S3 中托管我们的应用，其存储费用可以忽略不计（每月不到 1 美分），并且不会随着用户的增加而增加。

数据成本

应用加载之后，某些用户将希望保存他们的答案。要做到这一点，需要使用 Cognito 进行连接。就我们的应用而言，Cognito 是一项免费服务。只有用到 Cognito Sync（一个用来在客户端之间同步数据的功能）才会产生费用，但我们的应用没有使用。

使用从 Cognito 获取的安全凭证，应用将直接请求 DynamoDB 保存和获取用户的数据。与 S3 一样，Amazon 对 DynamoDB 中的数据传输也是收费的，但正如第 5 章“将数据存储在 DynamoDB 中”所述，它还会对读取和写入容量收费。由于我们提前购买了这个容量，因此能方便地控制在这项服务上的支出。

与网上的大多数事情一样，只有一小部分用户可能想要连接并保存他们的答案。如果保守假设只有 5%的用户会这样做，我们可以估算出 DynamoDB 必需的读写容量和成本。以下是 Amaon DynamoDB 服务的价格（截至 2016 年 1 月）：

10 次写容量单位	50 次读容量单位	传输成本（传出）	传输成本（传入）	存储成本（超过 25 GB 之后）
0.0065 美元/小时	0.0065 美元/小时	0.090 美元/GB	免费	0.25 美元/GB/月

根据我们假设的转换率，预计每 10,000 个用户中就有 500 个用户在我们的应用中连接 AWS 并保存答案。如果在应用中有 100 道题，并估计用户会答完所有问题，那么对于每 10,000 个用户，我们需要在账户的整个生命周期内执行 50,000 次写入，这并不多。换个说法，根据 DynamoDB 的 AWS 免费条件，我们每天可以免费执行高达 210 万次写入。

因为无服应用只使用它所需要的服务，所以我们的数据成本可以很低。与基于中间层应用服务器，将会话信息存储在数据库中的应用不同，我们的应用只在绝对必要时才访问数据库。随着应用的增长，可以随时配置读写容量单位，并且用完后就释放它们，按小时付费。一旦确定了应用在运行时需要的常规容量，就可以通过预付费的方式以更低的价格一次性购买一年或三年期的预留容量。

微服务成本

正如刚才看到的，了解 Amazon 服务的定价模式，我们才能从设计上降低应用的成本。为了更好地集成在第 6 章“使用 Lambda 构建（微）服务”一节创建的微服务，我们需要适应 Amazon Lambda 的约束和功能，然后调整应用，以达到性能和成本的精妙平衡。

Amazon 对 Lambda 服务按照每秒千兆字节收费。我们的函数其运行所需的内存和时间越多，要付的钱就越多。我们也按每个请求付费，虽然成本相当低，每百万次请求只需要

0.20 美元（而且每个月的前 100 万次请求是免费的）。我们将服务的大小和持续时间限制设置为 512 MB 和 5 秒，因此在这些限制条件下，可以计算出对于固定数量的请求运行服务的总成本最多能有多少。

截至 2016 年 1 月，Amazon 列出的 512 MB Lambda 函数执行的价格[11]，每 100 ms 为 0.000000834 美元，或每 5 秒 0.0000417 美元。Lambda 以 100 ms 为增量进行计费，因此一次成功的执行不大可能花这么多钱，但它给出的是最高的成本。这意味着在最坏的情况下，我们的服务执行 10,000 次将花费 0.417 美元。此外，Amazon 免费套餐可免费提供 400,000 GB 的执行。虽然这意味着我们的微服务必须非常受欢迎才会产生成本费用，但这一成本不包括我们需要购买以支持此服务的额外 DynamoDB 读取容量，这个数字将取决于我们拥有的数据量。那么，可以做些什么来降低这项服务的成本吗？

我们可能想到的第一件事是调整资源限制以适应函数的实际需要。5 秒是一个估计值，可能比这个函数真正需要的时间长。也就是说，如果一个函数运行了那么长时间，可能有问题，并且没有意义把钱花在一个将要失败的请求上。我们还可以通过更智能的服务实现来减少内存大小。我们的服务的第一个版本相当幼稚，使用 DynamoDB 的一些分页功能可能是一种让内存占用量随着用户数量的增长始终保持在较低水平的方法。

为了降低对读容量的需求，我们可能要求用户在使用此服务之前进行身份验证。如果它支持的功能是令人信服的，将通过限制请求的数量同时提高我们的转换率，并降低支持此服务的成本。当然，这么做是否适合这个应用（或别的应用）取决于应用的目标，但是为优质用户保存更昂贵的服务有时可能是降低成本的最简单的方法。

加起来

因此，假设每个月新增 85,000 个新用户，而我们的应用在第一年用户就超过 100 万。为了支持这些用户，我们将购买 25 个额外的写入和读取容量单位，为我们通过 Amazon 免费包获得的 25 个单位提供补充。根据之前的计算，这个值远远高于我们对 1 年内的写入量

11 https://aws.amazon.com/lambda/pricing/

的估计，但是还必须考虑到流量峰值时的情形。

要是根据平时的使用情况购买写容量，意味着你的应用有一半的时间都会崩溃。

接下来，我们需要支付 DynamoDB 的传输成本。Amazon 对这种数据传输的收费条件与对 S3 的相同，也是 0.09 美元/GB。但是，我们只需要为传出的数据付钱。发送到 DynamoDB 的数据是免费的。我们应用中的答案记录很小（不到 1 KB），因此即使每个接入的用户回答 100 道题，数据传输成本也几乎可以忽略不计。

对于我们的微服务，假设我们限制接入用户的使用，并且一位典型的接入用户在一年内访问该服务 10 次。这意味着在第 1 年预计将有 50 万次 Lambda 服务调用。由于每月的头 100 万次请求是免费的，并且我们对此服务的使用远未达到收费范围，因此不需要支付任何费用来支持前 100 万位用户。最后，由于我们使用 Cognito 和外部身份服务商来管理用户账户，也没有任何成本。

把所有这些成本和估算值放在一起，就可以预测如果有 100 万用户在第 1 年加载我们的应用，我们将花费多少钱。从下面的表中可以看出，总成本约为 195 美元，即每天 0.54 美元。

服　务	容　量	费　用
静态页面寄存	100 万次应用加载	23.64 美元
身份管理	50,000 个用户信息	免费
写容量	15 亿次写（最多）	142.35 美元
写数据传输	500 万次，1 KB 写（平均）	免费
读容量	15 亿次读（最多）	28.47 美元
读数据传输	500 万次，1 KB 读（平均）	0.43 美元
微服务	500,000 次执行	免费

当然，这只是一个估算值，实际成本将取决于人们对应用的使用情况，但这些数字是有意义的。我们知道即使有数百万的用户，每月的成本也是可以控制的。由于我们不必托管应用服务器层，因此节省了虚拟或物理服务器的成本，这些服务器不仅有运行成本（无

论是否使用它们），而且在横向扩展时还需要花很多时间进行管理。

为应用扩容从来都不容易，但是既然你了解了技术、工具和成本，也许觉得准备好构建一个无服务器的应用了。随着需求的增加，你对人们究竟如何使用应用所做的假设将得到检验，你可能会对用户做的事情感到惊讶。当然，有时这是一种惊喜，而有时可能会是惊吓。但是，构建无服应用可以让你拥有非常多的选择，快速而有效地应对挑战。你可以出奇招以处理意想不到的需求，并且为你适应情况争取时间。或者当使用模式发生变化时，你可以进行调整以节省资金。使用无服应用程序，只需片刻就可以在这两个极端之间随意切换。

再次部署

世上有两种软件：顺势而变的和死亡的。如果你的应用非常有用，很成功，你会需要修改它，为它添加新功能，使其支持更多用户和存储更多数据。从现有代码中挤出附加值是开发人员最有价值的技能之一。如果你擅长这个，可以一直做下去。

本书中所讲的方法将帮助你做这件事。随着应用程序的增长和变化，我们添加的测试将帮助你自信地一次又一次地部署，知道自己在一天、一个月或一年前新增的功能仍然有效。用于构建应用的简单工具将在未来几年仍然实用，我们使用的基础设施可以接纳新的用户和功能，同时保持低成本。

既然你已经掌握了本书中的工具和技巧，对下面这些主题你可能想做一番研究。

下一步

我可以复用此工作空间吗?

是的，如果你愿意，可以在你的项目中使用本书的预备工作空间。与本书的内容不同，这个工作空间是基于 MIT 许可的。但是，你可能需要更改 `sspa` 脚本以安置你

的应用。

创建客户端日志 Web 服务

通过创建使用 CloudWatch API 的 Lambda Web 服务，你可以直接从 Web 应用写入日志消息。为 `window.onerror`[12]附加一个处理函数，你就能捕获任何 JavaScript 错误，并将其发送给 CloudWatch 以供分析。你甚至可以创建警报，就像我们在本章前面做的一样，让自己知道用户何时遇到问题。

请记住，在执行安全审核时，不能信任从浏览器获得的日志消息。消息可能是伪造的或者根本不存在，因为攻击者（和其他人）对浏览器中发生的一切拥有完全的控制。但是，当对合法问题排错时，这些日志可能很有用。

使用 CloudFront 走向全球

正如我们在本章看到的，在 Amazon 的 CloudFront 中提供应用是防止 DoS 攻击的好方法。CloudFront 还提供了比 S3 中更高级[13]的缓存机制。随着你的用户遍布全球，你可能希望了解 CloudFront 有哪些降低成本，并改善用户对应用的体验的方法。

12 https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers/onerror

13 http://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/Expiration.html

附录A
安装Node.js

Amazon Lambda 运行 Node.js 4.3.2，可以从 nodejs.org 的站点（https://nodejs.org/download/release/v4.3.2/）下载。不论通过何种方式来安装 Node.js，最好尽可能选择接近的版本。

安装 Node.js 运行时

根据你的操作系统，从以下方法中选择一种安装。

Linux

在*nix 环境下使用开源软件的一个好处，是随时可以从源代码编译安装。Node.js 也是一样。打开 nodejs.org 站点的下载链接，选择 32 位或者 64 位的 tar 包（取决于你的操作系统）。然后下载，解压，按照 README 文件里的步骤操作。

如果不想从源代码编译安装 Node.js，还有其他的几个选择。如果你在使用 Ubuntu 或者其他基于 Debian 的发行版，可以用 `apt` 安装 Node.js，但有几点要预先说明。首先，Lambda

使用的那个 Node.js 版本可能没有。在编写本书的时候，可以从公开仓库获得的最新版本是 v0.10.25，和 Lambda 的 Node.js 版本有明显差别。

另一个安装 Debian 包的问题是可执行文件名和通过源码安装的是不同的。Debian 包将 Node.js 解释器安装为 `nodejs`，而不是该命令的名字 `node`。这会导致 node 包（比如 Jasmine）出现问题，这些包依赖于通过该路径才能获得的 `node` 可执行文件。

如果你确实通过 Debian 包管理器安装了 Node.js，就需要修复这个问题。可以在 `/usr/bin` 目录下创建一个名字正确的 symlink。执行这些命令，一切都能搞定：

```
$ cd /usr/bin
$ sudo ln -s nodejs node
```

你还可以在 Red Hat 或者 CentOS 上通过 `yum` 安装 Node.js，但在此书编写之时，那些版本和 Lambda 的 Node.js 版本差距更大（v0.12 以上），所以不推荐。

OS X

如果你使用 OS X，可以从本附录开头的那个链接下载一个可执行的 Node.js 安装器。只要解压，遵照指示操作就可以。这是获得 Lambda 使用的指定版本 Node.js 的最直接途径。

你还可以通过 Homebrew 和 MacPorts 安装 Node.js，但安装的版本可能会和 Lambda 环境不一致。使用这种方式要小心。

Windows

正如 OS X 一样，下载 Node.js 安装包是获得 Lambda 上运行的那个版本的最直接方式。打开本附录开头的下载链接，然后一步步按照 Windows 安装向导操作。

管理多个 Node.js 版本

如果已经安装了 Node.js，但是版本不对，该怎么办呢？通过一个叫作 nvm 的工具[1]是有可能安装并使用多版本 Node.js 的。

使用 nvm 命令，只需要一条命令就可以在 Node.js 的不同版本间切换，甚至是安装它们。要安装 Lambda 使用的指定 Node.js 版本，可以运行命令：

```
$ nvm install v4.3.2
```

要使用它，可以运行命令：

```
$ nvm use v4.3.2
```

假如你需要使用不同的环境（比如 Lambda 和 EC2 的组合），或者如果你想要使用不同的 Node.js 版本给本地工具来测试 Lambda 函数，可以同时安装多个版本。

1 https://github.com/creationix/nvm

附录B

分配一个域名

S3 终端的 URL 地址对于人类而言理解起来很困难，所以如果你在使用它们访问应用，会希望有一个更好的域名来将应用分享给其他人。幸运的是，S3 让你通过一条 CNAME 记录轻松地将一个域名关联到 S3 存储桶上。

域名注册商应该会提供关于如何为域名创建 CNAME 记录的说明。你可能想为与我们的 S3 存储桶同名的域创建一条 CNAME 记录。一定要输入终端地址（而不是存储桶的名字）作为 CNAME 记录的值。终端地址包含了存储桶名和它所在的区域，如下所示：

```
learnjs.benrady.com.s3-website-us-east-1.amazonaws.com
```

你可以将一个像这样的 S3 终端地址映射到与存储桶名匹配的域或者子域上。如果这个 S3 存储桶是配置来提供静态 Web 内容的，将会处理被发送给这个终端的请求，就好像这些请求是被发送到原先的主机名一样。

一旦 CNAME 记录被创建，就可以使用 `dig` 命令来检查，确保它是正确的，即使这些改变还没有被复制到其他 DNS 服务器上。你只需要知道你的注册商的名字服务器。例如，下例演示了我如何检查我的 DNS 记录：

```
$ dig @ns1.hover.com learnjs.benrady.com +short
learnjs.benrady.com.s3-website-us-east-1.amazonaws.com.
```

注意，CNAME 记录末尾有一个点，这是 DNS 要求的。如果你的注册商没有为你添加这个点，你可能需要编辑记录来自行添加。一旦这个 DNS 改动传播开，在所有浏览器中应当都可以看到应用。

创建 DNS 记录

DNS 即域名系统（Domain Name System），也就是你的计算机如何解释一个 URL 的主机名，把它转换为它可以真正连接的 IP 地址的方式。互联网上所有类型的协议都要用到 DNS，包括 HTTP、FTP、SSH 和邮件协议，比如 SMTP 和 IMAP。有很多种不同类型的 DNS 记录，需要哪一种记录取决于你想要做什么。域名经常会有一个 A 记录指向一个互联网上的一个 IP 地址。由于 Amazon S3 可能会随时修改存储桶的 IP 地址，不可能依靠那个 IP 地址来创建 A 记录。对于这个应用来说，你希望能创建一条 CNAME 记录，它将一个域名映射到另一个域名上。

每条 DNS 记录还有一个存活时间（TTL），它决定了多久从你的注册商名字服务器上刷新此条记录。这个时间范围可以是从一分钟到一天，所以 DNS 记录所做的修改可能需要一些时间才能传播到网络中所有的 DNS 服务器上。在这些改动传播完成之前，不能查看你的应用。你可以通过清除系统 DNS 缓存来加速这个过程。

参考书目

[Hav14] Marijn Haverbeke. *Eloquent JavaScript: A Modern Introduction to Programming*. No Starch Press, San Francisco, CA, second, 2014.

[How14] Shay Howe. *Learn to Code HTML and CSS: Develop and Style Websites*. New Riders Press, Upper Saddle River, NJ, 2014.

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

Serverless架构
无服务器单页应用开发

Serverless 架构是尝试新创意、探索可能的新市场或者创建最小可行产品的极佳应用开发方式。搭建这类应用来替代传统 Web 应用或者原生应用，意味着你可以更快接触到客户。你可以在若干小时内搭建一个初始版本并在几秒内部署。这些应用可以立即更新，轻松地被拆分测试，可以提供详细的使用指标，帮助你理解客户需求。

对于创业者以及中小企业的开发者来说，这是一个值得了解和学习的新方法。

Serverless是什么意思？

Serverless 的意思就是，你开发应用时可以专注于实现应用的业务逻辑，不需要考虑管理服务器的事情。采用 Serverless 架构，你的应用可以搭建在 Web 服务之上，而不是运行在需要你进行配置和管理的服务器之上。

这么说，Serverless应用的唯一好处就是我不再需要兼着做系统管理员的事情了？

非也。Serverless 应用还有很多其他的优点：扩展性好、可靠性强，而且成本低。把应用构建在 AWS 上，就算用户达到数百万，其运行成本每天也不过几毛钱。当你的应用规模增长时，通过 AWS 可以很方便地调配更多资源，而不必因为扩容进行重构。如果用户不多，你也只需要为必需的资源付钱，在很多情况下，运行这样的应用成本基本上为零。

Serverless应用与传统Web应用相比有什么优缺点？

传统 Web 应用一般使用 MVC 框架构建，要有应用服务器，而且大多数应用逻辑都放在服务器上。这一类 Web 应用不过是其服务器之上的一个接口，负责应用的所有核心功能：存储和处理数据、发布安全证书以及存放核心应用逻辑。而 Serverless 单页应用则把大部分逻辑放到浏览器中。这样做不仅统一了应用，而且更易于与 AWS 提供的高扩展和高可靠性服务集成。你完全可以依靠 AWS 的工程师实现应用的扩容，不再需要对负载均衡应用服务器做水平扩容。

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

责任编辑：许　艳
封面设计：吴海燕

上架建议：App开发

ISBN 978-7-121-31736-1

定价：65.00元